WINTER 1983

An Argus Specialist Publication

Personal SOFTWARE

£1.95

PLAYING YOUR WAY INTO BASIC

Learn about BASIC by playing games

Beginners' introduction to BASIC

Interpreters and Compilers explained

Pages of BASIC games to play

Use BASIC to write your own games

Advanced course for elegant programming

Learning BASIC is fun!

No 3 An Argus Specialist Publication Sept/Oct issue 1983

SPECTRUM COMPUTING

£2.99

The original on-screen computer magazine!

Compatible with both 16K and 48K machines

Over 80K of program! Simply LOAD & RUN

3D Special issue!

3D construction program for you to build your own creations

3D strategy air combat game to run and play Ground attack – avoid the Rapiers

Machine-code pixel scroll to improve your graphics

Protect your colour and mix up some new shades! – new routines

On-screen reviews for:
Melbourne House
Ultimate
Imagine
Postern

SPECTRUM COMPUTING ISSUE NO 3
ASP SOFTWARE
145 CHARING CROSS ROAD, LONDON WC2H 0EE
© 1983

OUT NOW

WINTER 1983
An Argus Specialist Publication
Personal
SOFTWARE
PLAYING YOUR WAY INTO BASIC
Learn about BASIC by playing games
BASIC
Learning is fun!

Volume 2
No 3
Winter 1983

Editor: Wendy J Palmer
Divisional Advertisment Manager:
Beverley McNeill
Senior Sales Executive:
David Poulter
Advertisement Copy Control:
Ann McDermott
Managing Editor:
Ron Harris BSc
Chief Executive:
T J Connell

Origination and design by MM Design & Print

Personal Software is a quarterly magazine appearing on the third Friday in May, August, November and February.

Distribution by: Argus Press Sales & Distribution Ltd, 12-18 Paul Street, London EC2A 4JS Tel: 01-247 8233. Printing by: Alabaster Passmore & Sons Ltd, Tovil, Maidstone, Kent.

The contents of this publication including all articles, designs, plans, drawings and programs and all copyright and other intellectual property rights therein belong to Argus Specialist Publications Limited. All rights conferred by the Law of Copyright and other intellectual property rights and by virtue of international copyright conventions are specifically reserved to Argus Specialist Publications Limited and any reproduction requires the prior written consent of the Company. © 1983 Argus Specialist Publications Limited.

Subscription Rates : UK £7.80 for four issues including postage. Airmail and other rates upon application to Personal Software Subscriptions, 513 London Road, Thornton Heath, Surrey CR4 6AR.

CONTENTS

Editorial & Advertisement Office
145 Charing Cross Road, London WC2H 0EE
Telephone: 01-437 1002 Telex: 8811896

Preface 4

Beginning BASIC – 1 5
Our series on teaching BASIC starts with algorithms and flow charts.

Beginning Basic – 2 8
We take a look at some of the more common BASIC statements.

Beginning BASIC – 3 11
Conditional branching instructions are a vital part of BASIC.

Beginning BASIC – 4 13
PRINT instructions and more.

Beginning BASIC – 5 16
INPUT and READ statements explained.

Beginning BASIC – 6 20
Match your wits against the computer with the game of NIM.

Beginning BASIC – 7 24
Find the winning strategy for NIM.

Beginning BASIC – 8 28
An introduction to Extended BASIC.

Beginning BASIC – 9 31
Don't let your computer STRING you along.

Beginning BASIC – 10 34
Fathom the mysteries of hexadecimals.

Beginning BASIC – 11 37
We take a look at binary searches.

Beginning BASIC – 12 40
The concluding article in the series looks at sorting.

Sniper 42
Outwit the malevolent robots on your Genie or TRS-80.

Sardaukar Assault 45
Save the universe from the Sardaukar fleet with your Atari.

Spectrum Zap 47
Zap the aliens using your ZX Spectrum.

Puzzle Square 50
A general BASIC game for rearranging squares.

Serpents 52
A classic snake-chasing game for the BBC Micro.

Interpreters 55
How is BASIC understood by your micro?

Compilers 58
We look at a commercial compiler.

Elegant Programming – 1 63
Now that you've covered the fundamentals of BASIC, we take a more advanced look at the language. We start here with the difference between programming and coding.

Elegant Programming – 2 67
We introduce the idea of structured programming.

Elegant Programming – 3 71
It's bug-hunt time as we look at crashproofing your programs.

Elegant Programming – 4 75
We now progress into the world of uncertainty and randomness.

Elegant Programming – 5 79
Writing structured programs requires the use of correct data structures.

Elegant Programming – 6 83
Data types are discussed along with the methods that can be used to structure them.

Elegant Programming – 7 87
Step into the world of graphics — it's really moving stuff!

Elegant Programming – 8 91
Sorting and searching lists of data made a bit easier.

Elegant Programming – 9 96
For our finale we look at a complete method of tackling problems in programming

Credits: Our grateful thanks to the following individuals for their assistance in the production of this issue. Jordan Bennet, Paul, Bill, Nigel, William, Gary, Roy, Mark, Adrian, Tom and others almost too numerous to mention.

PREFACE

Example
For the equation

When I was at school trying my hand at languages (as well as the odd bit of maths, science, etc) the range available extended from French, English and German to that quaintest of tongues, Latin. For the modern generation at school the range has been extended even further to include such delights as BASIC, FORTH and LOGO.

There is no point in owning a microcomputer if you do not understand the language that it uses. I say no point, but I am of course assuming that you will want to do more with your micro than merely buy a commercial tape or disc, plug in and then walk away after playing the game or collecting some results. I assume that you're interested in how the micro does what it does and that you would like to influence that, or direct it completely.

BASIC is the most common language that micros use, although it does vary depending on the micro (remember that one micro's BASIC is another micro's error message). Many people are frightened by the jargon that so often accompanies literature on microcomputing and we have thus compiled this issue of *Personal Software* in order to help explain what BASIC is, what it can do and how you can use it.

The series 'Beginning BASIC' introduces BASIC to the absolute novice, explaining some of the more common commands and showing how the language can be built up. 'Elegant Programming' goes much further and delves into the realms of structured programming and some more advanced ways of using BASIC.

We have also included articles on interpreters and compilers to help clear some of the mystery surrounding these 'buzz words'. As examples of what you can do once you've learned some of the 'basics' (groan!) we have brought together some of the games in BASIC that have been published in the magazine *Computing Today* and *Personal Computing Today*. In this way you should be able to look at the programs in a new light — try to figure out how and why the games were written in the way they were, and see if you could alter them to include different routines, make them play faster etc.

We have tried to ensure that the programs are error free, so if you find you are having problems with any of them, please check them again thoroughly or get a friend to help. If as a last resort you can still not get the game playing properly you may write to *Personal Software* with a SAE. Please note however that we can not answer any telephone enquiries.

Phil Cornes

BEGINNING BASIC – 1

Here is the first part of our software teaching series.

It, is, unfortunately, very easy when watching a computer in action to subconsciously endow the machine with intelligence — under no circumstances is this the case.

Regardless of whether you are programming in the simplest of machine codes or the most sophisticated of high level languages, there is no way that the computer can do anything other than what it has been programmed to do, and the signs of intelligence that we seem to detect are present only because of the skill of the programmer. In fact, programming today is becoming quite a major business area, simply because of the amount of skill involved. As with every other trade, however, there are various tools which are at the disposal of the programmer to help in his work — one of the most important of these being the flow chart.

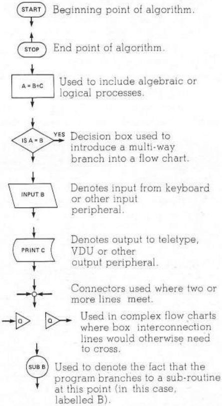
It does not matter what language we program in, be it machine code or BASIC, the technique of drawing and using flow charts is always the same.

We start with a problem, find an algorithm (finding an algorithm for a problem means finding a method of giving a complete and correct solution to the problem in a finite number of steps) to solve the problem, draw the flow chart and then write the program from the flow chart. In order that one programmer can understand another's work, certain conventions are adopted when drawing flow charts (see Fig. 1).

As a first example of algorithm and flow chart drawing, we will take the case of a young person applying for membership of a Social Club, wishing to discover what fees are payable as an annual subscription.

Consider the following — "The annual subscription for a man is £10, unless he is under the age of 25, when the subscription shall be halved. The annual subscription for a woman shall be £8, unless she is under 25, when the subscription shall be halved. Married women applying for membership shall be charged half the amount payable by a single woman over 25."

Fig. 1. Flowcharting symbols.



Beginning point of algorithm.

End point of algorithm.

Used to include algebraic or logical processes.

Decision box used to introduce a multi-way branch into a flow chart.

Denotes input from keyboard or other input peripheral.

Denotes output to teletype, VDU or other output peripheral.

Connectors used where two or more lines meet.

Used in complex flow charts where box interconnection lines would otherwise need to cross.

Used to denote the fact that the program branches to a sub-routine at this point (in this case, labelled B).

In this instance, it is unnecessary to find an algorithm to solve the problem as we are only going to use a flow chart as a means of simplifying the wealth of information given above (see Fig. 2).

So, for example, if you are a married female, it takes only a moment's glance at Fig. 2 to answer the questions "Are you a man?" (no) and "are you married?" (yes) to arrive at the knowledge that your annual subscription shall be £4.

You can see from this example how the flow chart helps to clarify and simplify an otherwise apparently complicated problem.

We will now go on to consider the generation of an algorithm, and to see how a flow chart can be drawn once an algorithm has been obtained. As an example, we will look at how it might be possible to get a computer to generate a representation of, and randomly shuffle, a pack of cards.

The first thing we need to do is to decide what would be an acceptable representation of the pack. We could reasonably consider the problem solved if the computer could be made to generate a list of the numbers 1 to 52 in a random order, so that each number from 1 to 52 would represent a different card.

The first method that springs to mind is to get the computer to open a set of 52 storage locations. The first random number between 1 and 52 can then be generated and placed in storage location number 1

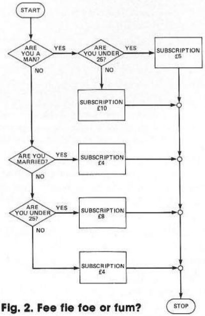


Fig. 2. Fee fie foe or fum?

(the method used to generate the random numbers is unimportant as far as the flow chart is concerned). A second random number is then generated and placed in storage location number 2, a third number in storage location 3, and so on until all 52 storage locations have been filled.

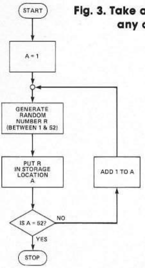
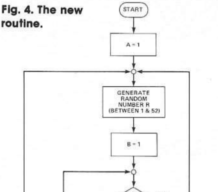

START
A=1
GENERATE RANDOM NUMBER R (BETWEEN 1 & 52)
PUT R IN STORAGE LOCATION A
ADD 1 TO A
IS A = 52?
NO
YES
STOP

Fig. 3. Take a card, any card...

Fig. 3 shows a flow chart to describe this algorithm. That appeared quite simple, didn't it? But if we give the problem some further consideration, you will see it is possible, since the numbers we are generating are random, to have generated two numbers which are the same. Indeed, this is most likely. This would mean that we would have at least two cards the same within one pack, and so our algorithm must be considered incomplete (though on the right track). To make the algorithm work correctly, we will have to include some form of check to ensure that when a number is generated which has already been used, it is not included in the list (see Fig. 4 for a flow chart which takes this point into acount). If you look through Fig. 4, you will see that a number is generated and then a check is made through all the storage locations that have already been filled to see if the number we have just generated has occurred before. If it has, then the number is ignored and a new random number is generated and checked; if it has not, then it is inserted into the next empty storage location. We then jump back and generate another random number and the process continues until all 52 storage locations have been filled.

This algorithm and subsequent flow chart would appear to be quite sufficient to solve the problem. But let us now consider this flow chart converted into a program and being run on a computer. Remember, every operation the computer executes takes some finite time to perform, albeit small, so that the more operations that need to be performed, the longer the program will take to run. This may appear to have been an obvious statement, but let us take a look now at our algorithm, bearing this point in mind. When we start off, with all storage locations empty, the first number we generate can be guaranteed not to have occurred before (though looking at the flow chart you will see that the computer does not know this) and can therefore be inserted straight into the first storage location.

Fig. 4. The new routine.



As the program proceeds, however, and more storage locations filled, it becomes more and more likely that the generated random number will, after some considerable checking, have to be abandoned and re-generated, until, when there are only two or three locations left to fill, we may have to generate and extensively check many tens of numbers to find one of the few remaining acceptable numbers. If the computer was made to print out each number as it was generated, we would notice a longer and longer time interval elapsing between the generation of consecutive numbers. Problems like this occur frequently when converting algorithms, where a solution which initially appeared to be satisfactory turns out to have some practical difficulties associated with it on closer inspection.

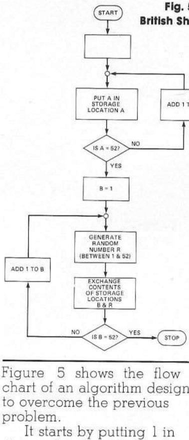

START
PUT A IN STORAGE LOCATION A
ADD 1 TO A
IS A = 52?
NO
YES
B = 1
GENERATE RANDOM NUMBER R (BETWEEN 1 & 52)
ADD 1 TO B
EXCHANGE CONTENTS OF STORAGE LOCATIONS B & R
NO
IS B = 52?
YES
STOP

Fig. 5. The British Shuffle?

Figure 5 shows the flow chart of an algorithm designed to overcome the previous problem.

It starts by putting 1 in storage locations 1; 2 in location 2; 3 in location 3; and so on until all 52 locations are filled, which in effect lays the cards out in sequence through the pack. It then takes the first location and exchanges its contents with the contents of another randomly chosen location, then the contents of location 2 are exchanged with the contents of a second randomly chosen location; the contents of location 3 are then exchanged with the contents of a third randomly chosen location, and so on until the contents of all 52 storage locations have been randomly exchanged in this manner. You may be a little sceptical as to whether the pack of cards thus generated was truly random. Experiments have, however, convinced us that it is. As you can see, there is never any need to generate more than 52 random numbers, because whatever the number generated turns out to be, we are always garanteed to use it, as it does not matter whether it has been generated before or not. Converting both of these flow charts into programs and running them on a computer, we discovered that this latter algorithm ran approximately ten times as fast, on average, as the first algorithm, so that there is a great saving in computer time used.

Looking through the algorithms and flow charts, you should begin to see that every operation a computer performs has to be very carefully planned and mapped out if a worthwhile program is to result. Although able to operate at extremely high speeds, the computer is merely manipulating pulses of electrical current according to a set of rules which the programmer lays down which, by careful manipulation and interpretation, can be made to have meaning.

In the next article, we will go on to consider the high-level programming language, BASIC, but do not forget the above routines, for when we have learned sufficient BASIC, we will be returning to look at them again and see how they can be implemented.

Would you like to pick a card?

YOUR MICRO COULD TEACH YOU A THING OR TWO ABOUT THE FRENCH...
...OR THE GERMANS...OR THE SPANISH

A home computer is an expensive toy; and, if playing games is all you do with it, a toy is all it is.

Now, using the New Personal Computer Superlearning System (PCSS) you can have fun with your micro and learn something at the same time.

PCSS language courses comprise 12 lessons on 3 audio cassettes used in conjunction with a fourth software cassette, to add a new dimension to learning.

Initially the software package enables you to see the words you're learning; then, as your vocabulary develops, it will test your skill in your new language.

Anyone can learn this way – no previous knowledge of the language is required. The unique PCSS method develops your overall learning and memory skills in a way thats both relaxing and enjoyable.

Each PCSS language pack – French, German or Spanish – contains a comprehensive booklet detailing the 12 audio lessons and the function of the interactive software. Additionally the booklet expands on the broader benefits of the PCSS method.

At only £29.95 per pack PCSS costs less than other home language courses yet it offers much more in terms of education and enjoyment.

Complete the coupon below and try PCSS for yourself – you'll be amazed what your micro can teach you.

MDA
COMPUTERISED EDUCATION SYSTEMS

(PCSS software is compatible with the ZX81 (16K), ZX Spectrum, BBC Micro, Acorn Elektron Micros.)
Each pack comes with a full money back guarantee if not completely satisfied.

Send your cheque or Postal Order for **£29.95** made payable to:
MDA Modon Associates Limited, 561 Upper Richmond Road West, London SW14 7ED.
or, alternatively telephone **Teledata 01 200 0200** and quote your Visa, Diners Club, Access or American Express number.

Tick which Audio/software package you require. (Prices include VAT. Add £1.45 for postage and packing on each order.)

Please supply the following Audio/software Packages

FRENCH ☐ GERMAN ☐ SPANISH ☐

Name:

Address:

Machine Type: Memory Size:

Phil Cornes

BEGINNING BASIC – 2

We take a look at some of the more common BASIC statements.

There are several dialects of BASIC in service at present, depending upon which machine you are programming for, and so we will make a start by looking at those parts of the BASIC language which are more or less universal.

The first thing to do is to define a few terms which we will be using throughout the rest of the series.

VARIABLE

A variable is a character (or sometimes group of characters) to which a numerical value may be assigned. The most common variable names and those which are used by most machines are the 26 single letters of the alphabet (A to Z). Other examples of variable names will be pointed out as we come across them.

ARITHMETIC OPERATOR

There are five common arithmetic operators in BASIC.

'=' — to be replaced by the value of (read as 'equals') equality operator
'+' — addition operator
'—' — subtraction operator
'*' — multiplication operator
'/' — division operator

Some machines have a sixth arithmetic operator '↑' which means 'raised to the power of' (may also be written as '**').

COMPARISON OPERATORS

There are six common comparison operators.

'=' — equals
'<' — less than
'<' — greater than
'<=' — less than or equal to
'<>' — not equal to (can also be '≠')
'>=' — greater than or equal to.

LOGICAL OPERATORS

There are two common logical operators.

AND — logical AND (may be written as *)
OR — logical OR (may be written as +)

Some machines also have a third logical operator — logical NOT (may be written as '~' or as 'NOT').

COMMAND

'Command' is the name given to keywords in BASIC which are used outside programs such as RUN, LIST, NEW etc. We will look at these in more detail later.

STATEMENT

The single instructions which go towards making up a program in BASIC are each called statements.

EXPRESSION

An expression is a collection of variables and/or numbers joined together by one or more arithmetic operators (so that 3*X+4, A—2 and A*(A+B)/(2—C) are all examples of expressions).

EQUATION

An equation is formed when an expression is assigned to a variable (so that Y=3*x+4, B=A—2 and Q=A*(A+B)/(2—C) are all examples of equations).

There is one other thing which should be discussed at this time. It can be described as follows—

If we let X take a value of 2 then what value do you think will be assigned to Y in the following equation:

Y=3+X*X.

If you thought the answer was anything other than 7 or 10, then you want to brush up on your maths. If you thought the answer was 7 then you are probably wondering where 10 came from and vice-versa. It all depends on whether you used a calculator or a computer to work it out. If you used a calculator then you would have worked it out like this—

Y=(3+X)*Y or Y=(3+2)*2

executing the operators as they occur and getting an answer of 10.

If, on the other hand, you used a computer, then you would have worked it out like this—

Y=3+(X*X) or Y=3+(2*2)

receiving an answer of 7. This may seem strange, but when computers do calculations they deal with the arithmetic operators in a certain order. First the computer scans the line left to right and performs all the multiplications and divisions as it encounters them; it then goes through again performing all the additions and subtractions that are left. The only way to alter the order of operations is to insert brackets where appropriate because the computer will work out the value of brackets before it does anything else and if there is more than one set of brackets one within another, it will work out the innermost brackets first.

So that, for example—

3+(2*(3+1*3))/(2+1)

has a value of 7 by the following reasoning.

The innermost brackets contain 3+ 1 * 3 which gives 6:3+ (1 * 3): moving out to the outermost brackets we multiply this by 2 to give 12. This is one partial solution. We then move on to the last pair of brackets containing 2+ 1 and evaluate this as 3. That takes care of all the brackets and gives an expression which looks like this—

3+ 12/3

Division now comes before addition and this reduces to—

3+ (12/3) or 3+ 4

then the addition is done to give a final answer of 7.

Try evaluating the following expression.

7+ ((7 * 8)/2(((12+ 8) * 2)/20)

When you have done this, try taking out all the unnecessary brackets (parentheses) without rearranging the order of the numbers (constants) and arithmetic operators so that the resulting expression gives the same result. The answers are given later as Fig. 6.

Certain facilities are required from any high level language, BASIC being no exception.

1. There must be a way of assigning values to any variables used in a program;
2. A method of outputting answers is also a must;
3. The language must have branching capabilities and in particular conditional branching must be provided;
4. Other facilities such as subroutines, string handling and some pre-defined functions are also useful and are usually provided.

LET

In BASIC the easiest way of defining a variable is to use a LET statement.

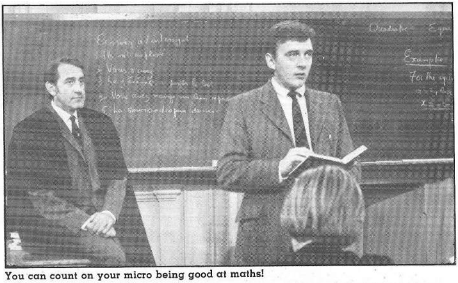
```
10 LET X= 3
20 LET Y=X—2
30 LET T=T+1
```

There are several things that you should notice here. Firstly, every instruction is preceded by a line number.

The computer, in executing a program in BASIC, does so in sequential line number order, starting with the lowest numbered line and going through to the highest numbered line except where a branching instruction is encountered in which case the next line number to be executed forms part of the instruction.

Secondly, the '=' sign in these instructions does not mean 'equals' in the normal sense of the word, but means 'to be replaced by the value of'. So that line 20, when translated into English, means something like this—

LET whatever exists now in the memory locations representing the variable Y 'be replaced by the value of' whatever exists now in the memory locations representing the variable X, minus 2.



You can count on your micro being good at maths!

This may seem a bit of a tongue twister (it would normally be read as "LET Y equal X minus 2") and has only been presented in this form to make line 30 a little easier to understand.

```
LET T=T+1
```

What does it mean?

Well, briefly, if the memory space for the variable T had a value of 2 before the execution of line 30, it would have a value of 3 after its execution. Got it? If not, refer back to the tongue twister, substituting the variables and constants from line 30 into line 20 where appropriate and read it through a couple of times until you have mastered this concept, because it is very important. You should now be able to see that where the following flow chart box appears in Fig. 3.

ADD 1 to A

this could now be replaced by a box containing the following BASIC statement—

LET A=A+1

It is reasonable to point out at this time that on most machines which can run BASIC the LET statement is optional so—

```
10 LET A=A+1
```

and

```
10 A=A+1
```

are equally valid statements so that where, in the last article, we encountered the following—

A=1

we were (apart from line numbers) already considering BASIC statements.

GOTO

The simplest form of branching ▶

instruction in BASIC is the GOTO statement, an example of its use is seen below—

```
10 LET Y=1
20 LET A(Y)=Y*Y
30 LET Y=Y+1
40 GOTO 20
```

The format of the GOTO statement is quite straightforward. The keyword GOTO is followed by the number of the line to which you wish control of the program to be transferred. Therefore, this program segment would be executed in the following order—

10 20 30 40 20 30 40 20 etc

As well as the GOTO statement, a new type of variable has been introduced in this program segment; the single subscript variable (can be known as a one dimensional array) represented here by the variable name A(Y). In Tiny BASIC as well as having single letters to represent variables (A, B, K, Y etc) you can also use variables of the following format—

A(1), A(2), A(10), A(50) etc

where A(1) is as different from A(2) as X is from Y. (The ETI Triton uses the @ symbol for its one single subscript variable, the TRS-80 Level I uses A).

Can we think of a use for these new variables? Well, if we think back to last month's card shuffling routines, we came across the following—

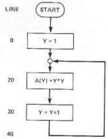
Put Random Number
R In Storage
Location A

where A took values from 1 to 52.

If we say that A(1) is storage location 1 and A(2) is storage location 2 or, in more general terms, A(A) is storage location A, then we have a representation for what was last time a set of 52 storage locations. So it is now obvious that the above flow chart box could be replaced by the following—

A(A)=R

Right! Let's go back to the program segment illustrating the GOTO statement.



This is its flow chart. We can see flow chart boxes representing lines 10, 20 and 30, but there is no box to represent the GOTO statement. It is merely represented by the box interconnection line which branches back to the connector between lines 10 and 20. While we are still on the subject, let's see what this program segment is actually doing (mentally executing a program without the aid of a computer is called DRY RUNNING a program).

Line 10 is the first line to be executed and all it is doing is assigning an initial value to the variable Y (in this case the value is 1). Now comes the line that might cause a bit of a problem.

```
20 LET A(Y)=Y*Y
```

If we think about the current value of the variable Y and substitute this value in the appropriate places, then what we end up with should make a lot more sense.

```
20 LET A(1)=1*1
```

All this says is "Write 1(1 * 1) into the memory space representing the variable A(1).

Now we pass on to line 30, which we have met previously, and this line adds 1 to the memory locations representing the variable Y; so Y now has the value 2.

Line 40 is the GOTO statement which tells us that the next line to be executed is line 20 again, and so we go back and re-write.

```
20 LET A(Y)=Y*Y
```

as

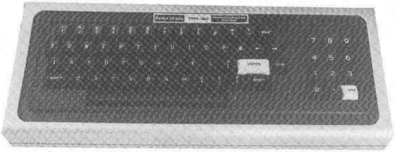
```
20 LET A(2)=2*2
```

using the new value of Y so that we write 4(2 * 2) into the memory space representing the variable A(2). This process is now repeated for variables A(3) A(4) A(5) and so on. Unfortunately, we have included no method of stopping the program or of branching out of this loop as we have not yet covered such things, but bear with us and all will be revealed.

You may have noticed from the explanation so far given that this program segment is calculating the points for a graph of $y=x^2$ and if we could look into the memory spaces representing the values of the variable A(Y) we would see the following—

A(1)... 1
A(2)... 4
A(3)... 9

One of the unfortunate points about this program is that it is an infinite loop (ie it will go on for ever with increasing values of Y) so we will now go on to look at a method of controlling the number of times we go round the loop.



The TRS-80 uses A for its single subscript variable.

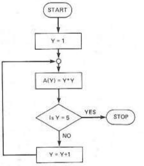
Phil Cornes

BEGINNING BASIC – 3

Continuing our look at BASIC, we investigate some of the conditional branching instructions featured in this language.

The first BASIC's really powerful conditional branching statements is IF THEN (we look at the others below) that go into the make-up of BASIC and we will add an IF THEN statement to the previous program segments to see what it can do.

Consider the following—



This is the same flow chart that we saw earlier except that now there is a two-way branch added which is made dependent upon the answer to the question 'IS Y = 5'.

Before we go on to look at the program derived from this flow chart, there is one other thing we need to consider. You will notice from the flowchart that IF Y is 5 when the decision box asks the question THEN we branch to a stop box. The statement in BASIC which causes the execution of a program to terminate is the END statement and you will find one of these in the program.

There is no statement in BASIC which corresponds to the start box on the flow chart (that is just presented for our information) and so the first box we consider contains Y = 1. The statement needed to convey this to the computer is—

```
LET Y = 1
```

but remember that every statement in a program must have a line number, and so we have—

```
10 LET Y = 1
```

We now move on to the next box

A(Y) = Y * Y and produce the statement—

```
20 LET A(Y) = Y * Y
```

The third box is the new one and we write—

```
30 IF Y = 5 THEN
```

THEN what? Well, we have to branch to the line number which contains the END statement, but we don't yet know which one this will be. So we can either sit and wait until we have written the END statement, or we can say always let the END statement exist on some high numbered line (say 9999) so if we ever need an END statement, we know what line it will appear on. We will do it this way so line 30 will read—

```
30 IF Y = 5 THEN 9999
```

and so if Y does equal 5, then we branch to the END statement that we will put in line 9999.

If the test (IS Y = 5) fails (answer is NO) then line 30 will be ignored and the computer will carry on executing the statements in the normal line number order.

The next box down contains Y = Y + 1, and so line 40 reads—

```
40 LET Y = Y + 1
```

From this we now branch back to the statement A(Y) = Y * Y which is on line 20 and we get—

```
50 GOTO 20
```

and lastly

```
9999 END
```

If we write this out in line number order, we get—

```
10 LET Y = 1
20 LET A(Y) = Y * Y
30 IF Y = 5 THEN 9999
40 Y = Y + 1
50 GOTO 20
9999 END
```

and this is our first complete program.

It does not matter that the line numbers do not follow on in multiples of 10, they don't have to, but what is more important is the fact that we do leave some numbers spare between our statements so that if we find we have missed out a line, or think of something else that we would like to add, then we have plenty of space to do so.

Consider the following—

```
 5 REM INITIALIZE Y
10 LET Y = 1
15 REM PUT Y * Y IN A(Y)
20 LET A(Y) = Y * Y
25 REM TEST FOR Y = 5
30 IF Y = 5 THEN 9999
40 LET Y = Y + 1
50 GOTO 20
9999 END
```

REM (I thought REM was an android or something to do with sleep) in BASIC is short for REMark and tells the computer that whatever follows on this line is to be ignored because they are only notes for the programmer as a reminder of what is happening.

REMark statements in a program of this length are unnecessary, but we will soon be writing programs of sufficient length and complexity to justify their use as memory aids.

Returning now to our IF-THEN statement (IF Y = 5 THEN 9999) the equals sign used here ▶

is not an arithmetic operator, but the first of the comparison operators. Any of the other comparison operators (<,>, = ,>= ,<>) could also be used in an IF THEN statement, so that—

```
30 IF Y>4 THEN 9999
130 IF Q<19.2 THEN 55
902 IF A(17)>= 14.9 THEN LET
      P=P+1
```

are all valid statements.

Notice here the twist in the tail of line 902. This is also a valid statement on most machines. This is easier to understand if we consider the IF-THEN statement as two separate statements. The first part (the IF part) asks a question (in line 902 — IS A(17)>14.9) to which the computer can answer either YES or NO. If the answer is NO then this statement is finished with and control passes on to the next higher numbered line. If the answer is YES then the computer passes on to the second statement on the line, the THEN part. THEN what? THEN LET P=P+1 or THEN END or THEN 900 (this is really an abbreviation of THEN GOTO 900) or THEN any other statement. We can even put another IF THEN statement in.

Consider the following—

```
200 IF (A= 1) THEN IF (B= 1)
      THEN 900
```

The computer encountering this would first ask the question IS A= 1. If the answer is NO control passes to the next higher numbered line, If, on the other hand, A is equal to 1, we move on to the statement following the THEN and encounter another IF THEN statement which is treated in exactly the same way as the first. IS B= 1. If NO then carry on with the next line, if YES THEN GOTO 900. You will see that using this logic we will only pass control to line 900 if both A= 1 AND B= 1. At about this point your memory should be stirring to the fact that you have read something about logical operators earlier and indeed this is the place where they fit in. Depending on which machine you are considering, there are two ways of re-writing line 200 above to achieve the same result.

You could use—

```
200 IF A= 1 AND B= 1 THEN
      900
```

which will normally be the format for machines with standard or extended BASIC, or—

```
200 IF (A= 11) *(B= 1) THEN
      900
```

for the tiny BASIC machines.

Notice the brackets in the second example. These tell the computer where one comparison ends and the other starts, otherwise the computer would attempt the following—

```
200 IF A= 1 *B
```

(multiplication sign!) and then bomb out on the second equals sign.

The other common logical operator (OR) can also be used in a similar manner—

```
300 IF Q>3 *H OR S<9 THEN
      R=R—2
```

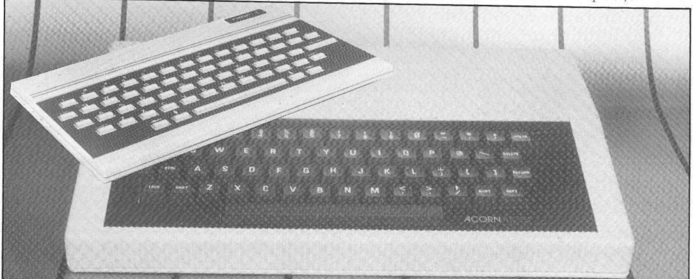
or

```
300 IF (Q>3H) + (S<9) THEN
      R=R—2
```

Notice the brackets again in the second example for similar reasons, and notice also the omission of the optional LET keyword before the R=R—2. We will continue to omit the LET from now on.

The answers to the questions posed in the issue are—

1. The expression has a value of 21, and
2. the expression could be simplified to 7+ 7 * 8/2/((12+ 8) * 2/20) You cannot remove the brackets round (12+ 8) * 2/20 (if you made this mistake, think about why not).



The Atom and the Oric both use BASIC but they can hardly be called the same!

Phil Cornes

BEGINNING BASIC — 4

We continue our series introducing the BASIC language.

We hope you got on all right with last time's homework; some sample answers and another question are presented at the end of this article. By now, some of you must be thinking it is all very well to be able to do vast amounts of calculation and decision making but as yet not a single answer has been printed out by the computer so that we can see the results of our labours. We will rectify this point straight away and go on to look at the main output of the BASIC language.

PRINT

The output statement of the BASIC language is the PRINT statement and an example of its use is given below—

```
10 Y=1
20 A(Y)=Y*Y
25 PRINT A(Y)
30 IF Y=5 THEN 9999
40 Y=Y+1
50 GOTO 20
9999 END
```

This you should recognise as our $Y = X^2$ program from the last article to which a PRINT statement has been added as line 25. Now line 20 calculates the value of A(Y).

The output from this program would be as follows—

```
1
4
9
16
25
```

each output being printed below the previous output in a vertical column.

We are going to spend some time on the PRINT statement as what you have just seen is the PRINT statement in only its simplest form, and it has several. Suppose, for example, that we wished to print a table with two columns, the first containing the value of X, the second the value of X^2 for values of X between 1 and 5. This is more or less what our program now does, except that values of X are not yet printed. If we replace line 25 with the following—

```
25 PRINT Y,A(Y)
```

then the output when the program was run would be—

```
1       1
2       4
3       9
4       16
5       25
```

which is just what we wanted. In effect, what happens is that the computer can split each output line on the VDU into sections (depending on the number of characters per line) about 16 characters long. Each of these is called a PRINT ZONE.
Whenever a new line is begun on the VDU, the first item to be printed starts at the beginning of the first zone on that line. When (as in our new line 25) a comma is encountered in a PRINT statement, it tells the computer that even though it has already output some data, there is more to follow on the same line so the computer will advance its cursor (the cursor points to the position on a line at which the next character will be printed) across the page to the beginning of the next totally empty print zone. The idea of "totally empty" print zone is brought in here because if you have print zones 16 characters wide and the output for the first print zone contains 18 characters (and so overflows into print zone 2) the next output will start at the beginning of print zone 3 as part of print zone 2 is already occupied.

There is another way of using the same idea as follows—

```
50 ........
60 Y=2
70 PRINT Y,
80 Y=3+Y+Y
90 PRINT Y
100 ........
```

Notice here the comma after the "Y" in the print statement of line 70. As we have already said, this tells the computer that there is more data to follow on the same output line in the next empty print one, so the output peripheral will wait while the calculation of line 80 is done then this result will be printed by the PRINT statement of line 90 alongside the first value of Y. One further point to note is that there is no comma following the "Y" in line 90, so if there is any subsequent output in the program it will now begin on a new line as the current line is finished with.

The next thing to note is that the PRINT statement has the ability to output messages as well as numerical answers so that, for example, you could get the computer to output—
TODAY IS 'WEDNESDAY'

The actual PRINT statement necessary to achieve this would have the following format — 50 PRINT "TODAY IS 'WEDNESDAY' "

You should be able to see from this example that the message has been enclosed within inverted commas in the PRINT statement, and these tell the computer to output whatever is between them direct and not to try to find a numerical value for it. The only character which cannot be placed within the inverted commas for printing is the inverted comma itself. Therefore, if you wish to put a quote into a PRINT statement, ▶

you have to use the apostrophe instead, as in the example given.

Consider the following:

```
20 ........
30 X=3*4/2
40 PRINT "THE VALUE OF 'X'
   IS",X
50 ........
```

Line 30 calculates a value for X, line 40 then goes on to PRINT the answer preceeded by the message (note the comma). The output looks like this—

```
THE VALUE OF 'X' IS     6
```

Notice the gap between the message and the answer. This arises because there are 19 characters in the message and with print zones 16 characters wide we just overflow into print zone 2 so the numerical value of X is printed in print zone 3. Ideally, we would like the output to appear as follows:

```
THE VALUE OF 'X' IS 6
```

and as you may have guessed this is possible on most machines by replacing the comma with a semi-colon—

```
40 PRINT "THE VALUE OF 'X'
   IS";X
```

The semi-colon has a similar effect to the comma in that it tells the computer that there is more output to follow on the same line, but the semi-colon differs in that it does not refer to print zones, but tells the computer to use close spacing between items to be printed (this can vary from 0 to 2 spaces depending on your machine).

FOR NEXT

It would be useful if there was an instruction in which we could state "execute this part of the program a number of times and then carry on with the rest of the program". Well (as you might have guessed from the sub-heading) there is such a statement in BASIC, the FOR NEXT statement which has the following general format.

```
FOR (variable)=(lower limit)TO
(upper limit)STEP(increment)
NEXT(value of variable)
```

For example—

```
10 FOR Y=2TO 6 STEP 1
20 ........
30 ........
40 ........
50 NEXT Y
60 ........
```

Here Y will take all values from 2 (lower limit) to 6 (upper limit) in steps of the increment (in this case, 1) so that the first time line 10 is executed, Y takes the value 2 and the program continues on until we reach line 50 (NEXT Y). We then go back to line 10 and STEP the variable by the increment and carry on to line 50 again. This looping continues until the value of the variable is greater than or equal to the value of the upper limit at which time the FOR NEXT loop is finished with. In our example, program execution would continue with line 60.

The lower limit, upper limit and increment can all be either constants, variables or expressions, so that—

```
FOR Q=A/B TO 19/C STEP R
NEXT Q
```

is a valid FOR NEXT statement.

One point worthy of note at this time is that there are two different ways of implementing a FOR NEXT loop on a computer, both of which are equally usable provided that you know which you have available.

Consider the following program—

```
10 FOR D=2 TO 1 STEP 1
20 PRINT "TEST IN NEXT
STATEMENT"
30 GOTO 9999
40 NEXT D
50 PRINT "TEST IN FOR
STATEMENT"
9999 END
```

which statements will be executed in the running of this program? There are two possibilities. They are—

10 20 30 9999

or

10 50 9999

Why the difference? Well, you should see if you look back that at some point in the execution of a FOR NEXT loop, we ask the question "is the value of the variable greater than or equal to the upper limit". If it is, then we have finished with the loop. if it isn't, then we go round the loop again.

The difference in the two executions is dependent upon whether we ask this question in the FOR statement or the NEXT statement.

Assume we ask the question in the FOR statement, then line 10 will make D equal to the lower limit (in this case 2) and we then compare this with the upper limit and find that it is already greater. So we have finished with the loop. If it isn't, loop. Control then branches to line 50 to PRINT the message, and ENDs in line 9999.

If, on the other hand, we ask the question in the NEXT statement, then line 10 assigns the lower limit value of 2 to D and control passes to line 20 which forms part of the loop. Even if our test would fail, we go through the loop at least once before we find this out. Many writers consider it to be a 'bad' interpreter that operates in this manner (test in NEXT statement), but in practice I have never encountered any difficulties, and I have found that most of the computers I have used do operate this way.

There is one other point we need to consider about the last program, and that is line 30. It is quite permissible in BASIC to interpret out of it to some other part of the program (not on a BEEB), but it is not permissible to branch into a FOR NEXT loop in such a way that the NEXT statement is encountered before the FOR statement: so that, if we were to add—

```
5 GOTO 40
```

to the above program, the computer would throw it out.

Another point to note is that

FOR NEXT loops can be nested one within another. For example—

```
10 FOR X=1 TO 5 STEP 1
20 FOR Y=1 TO 3 STEP 1
30 PRINT X*Y,
40 NEXT Y
50 PRINT
60 NEXT X
70 END
```

(Note that the FOR NEXT statement in Y is completely enclosed by the FOR NEXT statement in X. This is known as nesting.)

If you were to run this program, you would find that it would produce the following output—

```
1      2      3
2      4      6
3      6      9
4      8      12
5      10     15
```

a simple multiplication table.

Initially, lines 10 and 20 set X and Y to 1, line 30 multiples X and Y together and prints the result. (Notice the final comma in the PRINT statement). We then jump back to line 20 and increase Y to 2 and print the new value of X*Y alongside the first. This is repeated for Y=3.

When we hit line 40 for the third time, it is ignored, and we go on to execute line 50. All this does, in effect, is to close the print statement of line 30 so that the next output will start on a new line.

Line 60 now takes us back to line 10, where X is increased to 2. Line 20 (when entered from above) now resets Y to 1, and the whole process is repeated with values of Y=1, 2 and 3 again, and X=2, producing a second line of output. The third, fourth and fifth lines of output are then produced in the same way using values of X of 3, 4 and 5 and then the program ends.

Once you have a printer, you'll need to know how you want your output!

TAKE A CARD, ANY CARD

```
 5 DIM  A(52)
10 A=1
20 R=RND(52)
30 A(A)=R
40 IF  A=52 THEN END
50 A=A+1
60 GOTO 20
```

THE NEW ROUTINE

```
 5 DIM A(52)
10 A=1
20 R=RND(52)
30 B=1
40 IF  A(B)=R THEN 20
50 IF  B=A THEN 80
60 B=B+1
70 GOTO 40
80 A(A)=R
90 IF  A=52 THEN END
100 A=A+1
110 GOTO 20
```

THE BRITISH SHUFFLE

```
 5 DIM A(52)
10 A=1
20 A(A)=A
30 IF  A=52 THEN 60
40 A=A+1
50 GOTO 20
60 B=1
70 R=RND(52)
80 X=A(R)
90 A(R)=A(B)
100 A(B)=X
110 IF B=52 THEN END
120 B=B+1
130 GOTO 70
```

Figure 7

Try now to introduce FOR NEXT loops into the above program (The British Shuffle) so as to eliminate some of the IF THEN loops, and also get the program to PRINT out the cards it generates.

Phil Cornes

BEGINNING BASIC – 5

More ways of assigning a value to a variable.

You may remember, if you have been following the series, that we covered the LET statement earlier. This was the first of several statements that can be used to assign a value to a variable. Now we take a look at two more assignment statements.

INPUT

So far we have only seen the computer acting as an advanced calculator performing great amounts of arithmetic at fantastic speeds, but it can also be used as an interactive device, capable of asking questions and accepting answers. INPUT is the statement in BASIC which allows the computer to accept answers to questions.

Consider the following:

```
10 PRINT "INPUT AMOUNT (IN
POUNDS)"
20 INPUT P
30 V=P+(P*15/100)
40 PRINT "THE VALUE OF V
   IS",V
50 PRINT
60 PRINT "DO YOU WISH TO
INPUT MORE DATA"
70 PRINT "PLEASE TYPE 1
FOR YES OR 0 FOR NO"
80 INPUT A
90 IF A=0 THEN END
100 IF A=1 THEN 10
110 GOTO 70
```

This is a program which will add 15% VAT on to any amount of money that you INPUT in line 20. What happens is this:

Line 10 is a PRINT statement which is used here to give you an instruction. It is telling you to INPUT the amount that you want the VAT added to. Line 20 is the INPUT statement and when the computer encounters this line it will print a question mark on the screen (as a prompt to tell you that action is required) and then it will stop and wait for you to input a number from the keyboard (say 100) and enter it with a carriage return. When the computer has this value, it will be assigned to the variable which appears after the word INPUT (in our case P) so that when we reach line 30, P has the value 100 that we have just INPUT. Line 30 calculates the VAT and adds it on; line 40 prints the answer; line 50 prints a blank line; then line 60 asks if you wish to go through the process again with a new value of P and line 70 gives you the format you should use to make your reply; line 80 is the second INPUT statement which will take your answer from you and assign the value to the variable A; line 90 ends the program if you nswered 0 (no); line 100 branches you back to line 10 if you answered 0 (no); line 100 again, and line 110 branches you back to line 70 to reprint the answer format if your answer was anything other than 0 or 1. The flow chart for this program would appear as in Fig. 8.

It is possible to assign values to more than 1 variable in a single INPUT statement. The format of the INPUT statement would then be as follows:

```
100 INPUT P,Q,R,S
```

with each of these variable names speared by a comma. One thing to note, however, is that when you answer such an INPUT statement, you must enter as many numbers as there are variables requiring values, and each number you enter must be separated by a comma, so that:

```
10,20,30,40
```

would be an acceptable reply to the INPUT statement on line 100 above.

READ, DATA, RESTORE

Suppose now that as well as calculating 15% VAT we also wanted to calculate 10% and 12%, then the previous program could easily be modified with the addition of some more lines of calculation to provide the answers. There is, however, another method of achieving the same result, using only one line of calculation and going through it three times, once with each of the three different rates of VAT. To do this, we use READ and DATA statements.

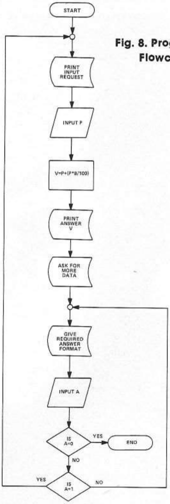
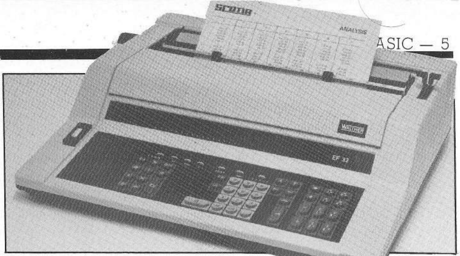


Fig. 8. Program Flowchart.

READ

The READ statement is very similar in format to the INPUT



Computers are not merely advanced calculators.

statement we have just considered, the difference being that when it is encountered, instead of stopping and waiting for the operator to enter a number, it takes the value to be assigned to the variable or variables from a DATA statement, which must appear somewhere in the program.

Consider the following:

```
10 FOR B=1 TO 3
20 DATA 1,2,3
30 READ X,Y
40 R=X*Y
50 PRINT X,Y,R
60 NEXT B
70 DATA 5,6,7
80 END
```

Line 10 tells us that we are going to loop round lines 20,30,40 and 50 three times: line 20 is a DATA statement which will be ignored by the computer; line 30 is the READ statement, and it tells the computer to assign a value to each of the two variables X and Y. The values to be assigned to the variables are contained to the two DATA statements in lines 20 and 70. The computer will start taking values from the lowest numbered DATA statement in the program (line 20) and assign the first value in that statement to the first variable in the READ statement, so here X would take the value 1. The second number in the DATA statement will be assigned to the variable Y, so Y would take the value 2. The calculation will then be done, giving an output from the PRINT statement on line 50 of:

```
 1          2          2
```

We then branch back to line 20 (with B equal to 2) which is again ignored, and so we encounter the READ statement again. The computer has already used the first two numbers of data and on the second reading will take the second pair of values (in this case the 3 on the end of line 20 will be assigned to X which finishes the DATA statement so that the computer will move up to the next higher numbered DATA statement in the program — line 70 — and the value 5 from line 70 will be assigned to Y) and a new line of output will be produced from these two values. Notice that line 30 will be executed three times so that there must be three pairs of values in the DATA statements, and if you count them you will find that this is so.

As you may have gathered from the sub-heading, the READ, DATA statement has one other facility — RESTORE.

If you imagine a pointer, stored in memory, to "remind" the computer which is the next piece of DATA to be READ, then the effect of the RESTORE statement is to return this pointer to the first piece of DATA in the lowest numbered DATA statement. The simple program below (infinite loop) gives an example of its use.

```
 5 INPUT A
10 FOR B=1 TO 3
12 READ C
16 PRINT A+(A*C/100)
20 NEXT B
30 PRINT
35 RESTORE
40 GOTO 5
60 DATA 15,10,12
80 END
```

Here, for each value of A input in line 5, we add on VAT using three different VAT rates, 15%, 10% and 12%.

A sample running of the program could look something line this:

```
?100
115      110      112
?150
172.5    165      168
?
```

etc. I'll leave you to work out the details (good practice).

GOSUB RETURN

GOSUB is the statement used in BASIC to branch to a subroutine. The general format of this instruction is:

```
10 GOSUB xxx
```

where xxx is the first line number of the subroutine. A subroutine would normally be used where a particular set of calculations or operations occur several times in a program, and this would save having to write out the whole operation every time it was to be used. The GOSUB instruction is very similar to the GOTO statement, except that before the branch is made, the line number containing the GOSUB statement is stored. After the computer has executed the subroutine, we make it encounter a RETURN statement. At this point, the computer retrieves the line number of the GOSUB instruction which called the subroutine, and branches control back to that point. The program then continues as normal.

Consider the following example:

```
 5 PRINT "INPUT CIRCLE
RADIUS"
10 INPUT R
20 D=R*R
30 GOSUB 500
40 PRINT "AREA IS",Q
50 D=2*R
60 GOSUB 500
70 PRINT "CIRCUMFERENCE
IS",Q
80 END
500 Q=D*3.14159
510 RETURN
999 END
```

▶

Notice that there are two GOSUB 500 instructions (one in line 30, one in line 60), but only one RETURN instruction in this program. As explained earlier, this is because the computer has stored the line number containing the GOSUB instruction so that when the RETURN instruction is executed (line 510) the computer "knows" which GOSUB instruction to RETURN to.

The subroutine in the above example is only multiplying by π and then returning control to the main program. Lines 5 to 40 use the subroutine to calculate the area of a circle from the radius you INPUT (line 10) and lines 50 to 70 use the subroutine to calculate the circle's circumference. One further point to note is the inclusion of line 80. This is most important. If we did not include an END statement at this point, then after the execution of the PRINT statement in line 70, the computer would crash into the subroutine by executing line 500, and then would bomb-out trying to execute line 510, as it has no RETURN line number stored. (See Fig. 9 for the flow chart of this program.)

FUNCTIONS

We will now take a look at some simple functions available in BASIC before we take a break and look at a game.

RND(X)

There are many variations of the RND function available on different machines but they all generate random numbers. The most common variation generates a decimal number in the range 0 to 1 (where 0 can be generated but 1 cannot). Another possibility is that the use supplies a number in brackets after the word RND and the function then returns an integer in the range 1 to the number in the brackets, eg you could simulate a dice with

```
D=RND(6)
```

As this is the simplest to use we will continue with it in future examples.

TAB(X)

This function is used in the PRINT statement and is very useful for spacing out headings and generating vertical columns of figures (if we want more than four columns so that we could not use the print zones). It is also useful for plotting graphs.

If you have every used a standard typewriter, then you should know all about TABS, eg:

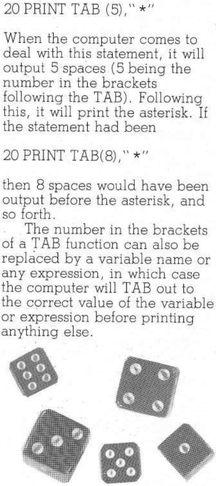
```
20 PRINT TAB (5)," *"
```

When the computer comes to deal with this statement, it will output 5 spaces (5 being the number in the brackets following the TAB). Following this, it will print the asterisk. If the statement had been

```
20 PRINT TAB(8)," *"
```

then 8 spaces would have been output before the asterisk, and so forth.

The number in the brackets of a TAB function can also be replaced by a variable name or any expression, in which case the computer will TAB out to the correct value of the variable or expression before printing anything else.



Consider the following:

```
10 FOR X= 1TO5
20 Y=X*X
30 PRINT TAB(Y)," *"
40 NEXT X
50 END
```

You will find, if you work through this program, that it will print a graph of $Y = X^2$ (rotated clockwise 90°). Try it, and see.

One last point about TAB is that the comma which follows is not interpreted by the computer as an instruction to move into the next print zone, but merely to separate the TAB function from anything which follows it.

ABS(X)

This function produces the absolute value (or modules) of the contents of the brackets (numbers, variables or expressions) which means that whatever sign (positive or negative) the contents of the brackets have now, the sign will be positive when the ABS function has been performed, so that—

```
10 X=ABS(T+3)
```

would have the value of + 1 if T was —4, or the value of + 4 if T was + 1, etc.

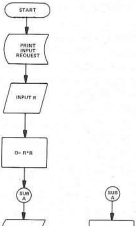
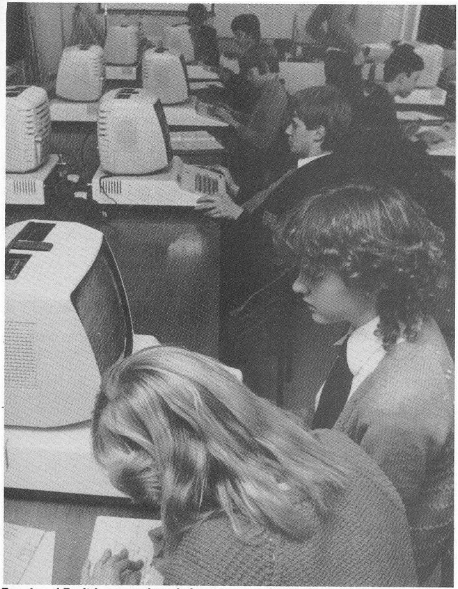


Fig. 9. Circumference program Flowchart.

INT(X)

This function is not applicable to computers operating with integer — only arithmetic for reasons that will become obvious in a moment. The brackets after the INT function can contain a number, variable or expression, and what the INT function does is to return the largest integer which is less



French and English are not the only languages taught in schools today.

than or equal to the contents of the brackets. This may sound something of a mouthful, but what it really means is that it makes positive numbers less positive, and negative numbers more negative.

For example:

X would take the value 2 if we executed:

```
10 X=INT(2.9)
```

and X would take the value —3 if we executed:

```
10 X=INT(—2.9)
```

There is one other statement that we need to consider in tiny BASIC before we go on to look at commands.

STOP

The STOP statement is similar in effect to the END statement, in that execution of the program ceases, but when a STOP statement is executed it also generates a print out, usually something like this:

```
BREAK AT xxx
```

where xxx is the number of the line which contained the STOP statement. This can be useful to "freeze" the display of a large table, for example, because one other facility of the STOP statement is the ability to start the program up again from the point at which it left off. To do this, we use the first of the commands.

CONT

This command is short for CONTinue, and is used to restart a program whose execution has been halted by means of the STOP statement considered above.

RUN

This is the command which is used in BASIC to cause execution of a program in memory to begin. Most computers will also accept RUN command in the following format:

```
RUN xxx
```

where xxx is the number of the line at which you wish execution of the program to begin.

NEW

This is the command in BASIC which tells the computer to erase the current program from memory to create memory space for a NEW program or other work.

LIST

This command causes the computer to LIST the current contents of the program memory and, like the RUN statement, it is also usual to be able to execute the command:

```
LIST xxx
```

where xxx is again the line number of the first line to be listed.

As a little exercise, think about the following problem. We will need the solution in the game which follows.

Given any number from 0 to 7, convert it to a 3-bit binary number such that each bit is assigned to one of three different variables. So that, for example, if the number was 7, the binary number 111 might have its three digits as follows:

```
H=1
J=1
K=1
```

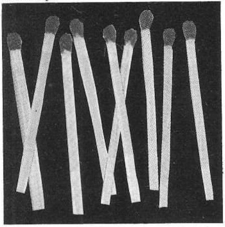
so that if we were to execute the statement PRINT H,J,K then the output would be the binary number.

Phil Cornes

BEGINNING BASIC – 6

In this article we examine an ancient game.

I can well imagine that the problem at the end of the last article might have given one or two of you quite a headache if you attemped it seriously. If it did, don't worry, because we shall we going into this problem in some depth as it forms a crucial part of the game program that we are going to develop.



NIM

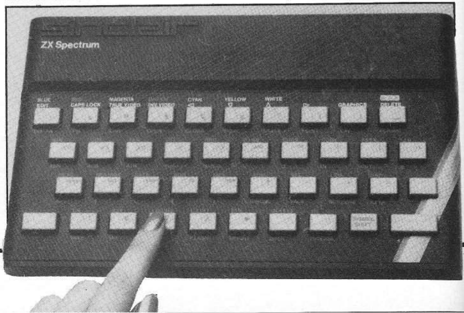
Many of you will already have played this game, indeed some of you may well be up to grand master standard. Nim is played with piles of matches. Two players take it in turn to remove matches from the piles until one of them takes the last match and is declared the winner. The computer versions of the this game can have anything from 3 to 6 piles with from 1 to 7 matches in each pile. The rule by which the players remove matches is as follows. Each player can remove as many matches as he likes on his turn, but from only one of the piles. For example, if a game were to begin with 3 piles each containing 7 matches, the first player's move would consist of choosing on of these piles and removing from it anything between one match and the whole pile. This rule applies right down to the end of the game, so that if at the end a single pile remains containing 3 matches the person whose turn it is to play can remove the whole pile and by doing so he also takes the last match to win. Though it has but a single rule, this game can be surprisingly subtle as you may well soon see.

Before we delve into the game proper, a sample answer to the problem could be as follows.

```
6014 INPUT Q
6015 RESTORE
6020 FOR T = 0 TO Q
6030 READ V,B,M
6040 NEXT T
6041 PRINT V,B,M
6042 GOTO 6014
7000 DATA 0,0,0,0,0,1,0,1,0,0,
1,1,1,0,0,1,0,1,1,1,0,1,1,1
```

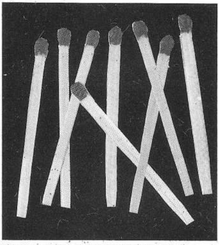
If you look at the program listing for NIM below, you will find that lines 6015, 6020, 6030, 6040 and 7000 are more or less the same as those given in the answer above, so you can see that the idea of converting decimal to binary is used in this program. The way that the above program segment works is as follows:

Line 6014 takes the value to be converted and assigns it to the variable Q. Line 6015 does nothing the first time it is executed, but we shall see why it is included later. Line 6020 sets up a FOR NEXT loop which will be executed the same number of times as the value of Q that was INPUT earlier. Line 6030 forms the contents of the FOR NEXT loop and what this does is to READ values for V, B and M from the DATA statement in line 7000. You should see, if you examine the DATA statement, that when we reach line 6040, V, B, and M have been set to the binary equivalent of the current value of T. We will go round the FOR NEXT loop until T equals the value of Q input in line 6014, by which time V, B, and M will be set to the binary equivalent of the value of Q. Line 6041 prints the values of V, B, and M that have just been "looked up" and line 6042 returns control to line 6014 to ask for another number to be converted. You should now be able to see why line 6015 has been included, because it now RESTORES that data pointer back to the beginning so that we can READ from the start of the DATA list again for the second and subsequent values of Q. I can hear you protesting that I have cheated by using this method of conversion, and I admit that it is not very efficient if there are large numbers to be converted, as the DATA statements would soon grow unwieldy. However, as the maximum number to be converted is 7, this method works quite well.



THIS ARTICLE'S EXERCISE

The operation and flow chart of the program listed below will be presented later because the computer uses a very precise mathematical method for calculating its moves, and if we were to go into the method now, there would be no point in playing the games as the outcome could be determined with certainty before the game was actually started. So all that will be presented for now is a full program listing which should be more or less self-explanatory in its use (see Fig. 10). The program has been written in as general a fashion as possible. However, one or two changes may be necessary as you type the program into your machines, because of the slighty differing facilities offered by various machines.

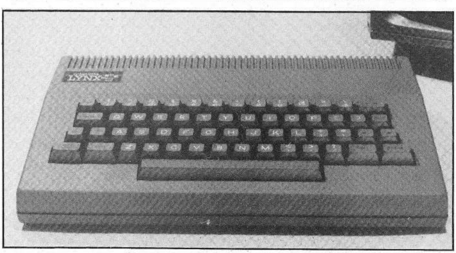


THE NIM PROGRAM

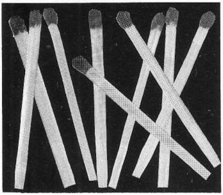
The first thing to note is that the listing of Fig. 10 is a program complete in itself. However, the program as presented in Fig. 10 does not give instructions on how to play the game, nor does it check the validity of the various inputs that you will be required to make furing the playing of the game. It is presented in this way so as to reduce its program memory requirement, but if you find that you have sufficient memory space available on your machine, you can also add the program lines listed as Fig. 11 which make the program more complete by giving instructons and making the aforementioned checks.

Fig. 11. The listing for the game instructions

```
10   PRINT "DO YOU WISH TO SET UP THE GAME"
11   PRINT "(1=YES 0=NO)"
12   INPUT A
15   F=0
16   R=0
17   P=1
18   O=0
20   IF A=0 THEN 150
30   PRINT "HOW MANY PILES DO YOU WANT (BETWEEN 3 & 6)"
40   INPUT A
50   PRINT "HOW MANY MATCHES IN PILE (BETWEEN 1 & 7)"
60   FOR X=1 TO A
70   PRINT X,
80   INPUT A[X]
90   NEXT X
100  GOSUB 5000
110  GOTO 170
150  A=INT(RND(0)*4+1)+2
153  FOR X=1 TO A
156  A[X]=INT(RND(0)*7+1)
160  NEXT X
163  GOSUB 5000
170  PRINT "DO YOU WISH TO START FIRST (1=YES 0=NO)"
175  INPUT S
180  IF S=1 THEN 2000
190  O=0
200  FOR P=1 TO A
205  IF A[P]=0 THEN 245
210  FOR R=1 TO A[P]
220  GOSUB 6000
230  IF O=1 THEN 300
240  NEXT R
245  NEXT P
250  P=INT(RND(0)*A+1)
255  IF A[P]=0 THEN 250
260  R=INT(RND(0)*A[P]+1)
300  PRINT "I'LL TAKE";R;"FROM PILE";P
305  A[P]=A[P]-R
330  GOSUB 5000
340  IF F=0 THEN 2000
350  PRINT "I WIN......."
360  GOTO 3000
2000  PRINT "WHICH PILE DO YOU WISH TO TAKE FROM"
2010  INPUT P
2020  PRINT "HOW MANY TO BE TAKEN"
2030  INPUT R
2035  A[P]=A[P]-R
2040  GOSUB 5000
2050  IF F=0 THEN 190
2060  PRINT "YOU WIN....."
3000  PRINT "DO YOU WISH TO PLAY AGAIN"
3001  PRINT "(0=NO 1=YES)"
3010  INPUT S
3020  IF S=1 THEN 10
3030  END
5000  Z=0
```



The next thing which might need changing is the use of the single subscript variable A [X]. The square brackets may be unacceptable to some machines, in which case they should be changed to the more conventional round brackets. Also, on some of the tiny BASIC machines, the only single subscript variable available is the @ array, so you will need to substitute this for the A array used in Fig. 10. Another problem may occur on tiny BASIC machines with statements like those on lines 150, 156, 250 etc., which are used to generate random numbers, as there is no INT function needed on an integer-only computer.



For example, line 150 could be changed to:

150 A = RND(4) + 2

For those of you who do not remember, there are two different types of RND function, one of which uses RND(O) to generate a four- or five-digit decimal number between 0 and 1 which then has to be converted by multiplications, additions and the use of the INT function to bring it within the desired range of random numbers. The other type of RND function is RND(X) — available on most common mini-BASIC-languages— which generates a random integer between 1 and X. So the new line 150 above generates an integer between 1 and 4 and then adds 2 to bring it within the range 3 to 6. This is then assigned to A. Line 150 in Fig. 1 performs the same function, but operates in the following manner.

The listing for the game instructions (continued)

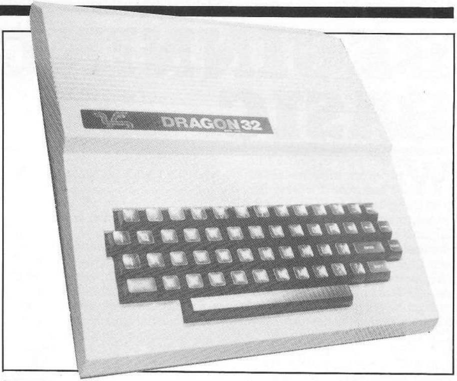
```
5010  PRINT "PILE NO.","NO. OF MATCHES"
5020  FOR Q=1 TO A
5030  PRINT Q,A[Q]
5040  IF A[Q]=0 THEN 5050
5045  Z=1
5050  NEXT Q
5051  PRINT
5052  PRINT
5060  IF Z=1 THEN 5080
5070  F=1
5080  RETURN
6000  A[P]=A[P]-R
6002  Z=0
6004  U=0
6006  I=0
6010  FOR Q=1 TO A
6012  IF A[Q]=0 THEN 6060
6015  RESTORE
6020  FOR T=1 TO A[Q]
6030  READ V,B,M
6040  NEXT T
6045  Z=Z+V
6050  U=U+B
6055  I=I+M
6060  NEXT Q
6070  A[P]=A[P]+R
6080  IF Z#INT(Z/2)*2 THEN 6120
6090  IF U#INT(U/2)*2 THEN 6120
6100  IF I#INT(I/2)*2 THEN 6120
6110  O=1
6120  RETURN
7000  DATA 0,0,1,0,1,0,0,1,1,1,0,0,1,0,1,1,1,0,1,1,1
9999  END
```

Fig.10. The complete NIM game program

```
5    GOSUB 9000
42   IF A<3 THEN 30
44   IF A>6 THEN 30
46   A=INT(A)
82   IF A[X]<1 THEN 70
84   IF A[X]>7 THEN 70
86   A[X]=INT(A[X])
2012 IF P<1 THEN 2000
2014 IF P>A THEN 2000
2016 P=INT(P)
2018 IF A[P]=0 THEN 2000
2032 IF R<1 THEN 2020
2034 IF R>A[P] THEN 2020
2036 R=INT(R)
9000 PRINT "DO YOU WANT INSTRUCTIONS (1=YES 0=NO)"
9010 INPUT D
9030 IF D=0 THEN 9200
9040 PRINT "THIS IS NIM"
9050 PRINT "THE GAME IS PLAYED BETWEEN ME AND YOU. WE PLAY WITH"
9060 PRINT "PILES OF MATCHES (BETWEEN 3 AND 6 PILES WITH UP TO"
9070 PRINT "7 MATCHES IN EACH PILE. THE OBJECT OF THE GAME IS "
9080 PRINT "TO TAKE THE LAST MATCH TO WIN."
9090 PRINT "THE RULE BY WHICH MATCHES ARE REMOVED FROM THE PILES"
9100 PRINT "IS AS FOLLOWS;"
9110 PRINT "WE TAKE IT IN TURNS TO MOVE AND ON EACH TURN YOU CAN"
9120 PRINT "TAKE AS MANY MATCHES AS YOU LIKE, BUT FROM ONLY ONE "
9130 PRINT "PILE IN ANY ONE MOVE."
9140 PRINT "WHEN WE START YOU WILL HAVE THE OPTION OF EITHER "
9150 PRINT "SETTING THE BOARD UP YOURSELF OR LETTING ME DO IT."
9160 PRINT "I WARN YOU, I AM GOOD AND SO TO GIVE YOU A CHANCE"
9170 PRINT "YOU ALSO GET THE OPTION OF WHETHER TO START FIRST"
9180 PRINT "OR NOT. I HOPE YOU ENJOY THIS GAME, AND HERE IS"
9190 PRINT "YOUR CHANCE TO FIND OUT."
9200 RETURN
9999 END
```

RND(O) is evaluated first, and generates a number in the range 0.00000 to 0.99999. This is then multiplied by 4 to give a number in the range 0.00000 to 3.99996. One is then added to this to give a range 1.00000 to 4.99996. The INT function is then applied to this value, which removes the decimal portion of the number to leave an integer between 1 and 4 (as in RND(4) above) and once again we add 2 to give a final range of 3 to 6 to be assigned to A.

Some machines may need the semi-colons in line 300 changing to commas.
The last point may need the semi-colons the line 6080 to 6100. The symbol "#" used in these lines means "is not equal to" and may need replacing with < > on some machines. Also, if you have an integer-only machine, you will need to replace these three lines as follows.



6080 IF Z Z/2*2 THEN 6120
and the same for 6090 and 6100 with the variables U and I. I hope you enjoy playing this game. We will come back to look at it in more detail soon. Now play the game and become familiar with it if you can and try to draw a flow chart and analyse how it works.

SO YOU THINK YOU'RE ROMMEL? ONLY £6.50 (plus 25p p&p)

INVASION places you in command of the land forces of the Western Alliance, just prior to an invasion by the Red's tank armies.

You will have to exercise skill and judgement as you move, supply and build up your limited defences to stand any hope of victory. Invasion is a true wargame and you could play all night just to find that the computer's grasp of strategy was superior to yours!

Using excellent graphics and with detailed instructions Strategy 1 is light years beyond the arcade. Isn't it time you stopped being the cannon fodder and took command? This is your chance to see if you're good enough to fight it out at the top!

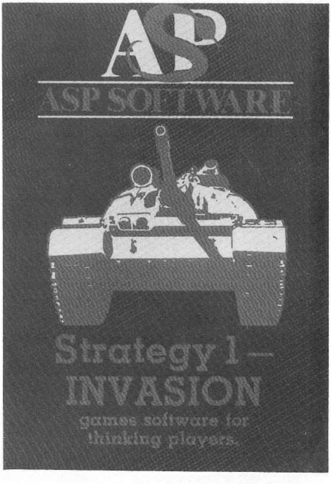
available for: ZX Spectrum (48K) and BBC Model B

Fill in the coupon and return it to: ASP Software
145 Charing Cross Road London WC2H OEE.

Please send me . . . . tape(s) – (delete as necessary) of Invasion . . . . . for. . . . . (state which version required). I enclose my cheque/ Postal Order/ International Money Order (delete as necessary) for: . . . . (made payable to ASP Ltd) OR Debit my Access/ Barclaycard (delete as necessary)

☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐

Please use BLOCK CAPITALS
Name (Mr/ Mrs/ Miss). . . . .
Address. . . . .
. . . .
Signature. . . . . Date. . . . .
Please allow 28 days for delivery



Phil Cornes

BEGINNING BASIC – 7

More details on the NIM game in the last article.

We will now examine this program in some detail to see how it works. The first thing to do is to look at the winning strategy as this, after all, is the strategy that the computer should adopt. The winning strategy for the game if NIM is quite well known, and can be found in several maths textbooks on games theory (try Methematical puzzles and Diversions by MARTIN GARDENER). Due to the lack of space, this strategy will only be stated and not derived.

WINNING COMBINATIONS

The first thing to do is to convert the number of matches in each pile into binary. As there are up to 7 matches in each pile, three binary digits are required for this conversion. The next thing to do is to add together all the first digits of the binary numbers produced, then add all the second digits together, and all the third digits. All these additions are done in decimal. When this is done, you are left with three decimal integer answers (see table 1). In our example, these are 4, 3 and 3. If any of these three digits is odd (which two of ours are) then the person next to play is in a winning position. The object now is for that player to remove some matches such that numbers are even so that, for example, removing 3 from pile 1 would leave the three digits as 4, 2 and 2 (see table 2) which is thus a losing position for the player whose it is to play next.

ALL IS REVEALED

Having looked briefly at the winning strategy which the computer adopts, let us now go on to look at the program presented as figure 10 last time. It would help if you could have that article in front of you as you read this description.

The program can very conveniently be broken into 5 main sections.

Section 1 is a subroutine which prints the current position of the board and also checks for a winning play. (Program lines 5000-5080. Flowchart Fig. 12).

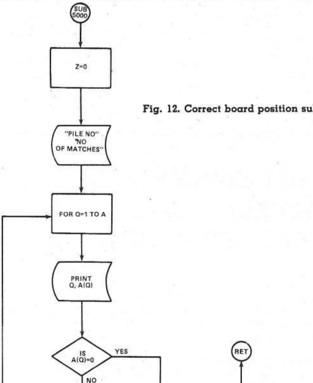


Fig. 12. Correct board position subroutine

Section 2 is also a subroutine and what this does is to convert the number of matches in each pile into binary and add them up as described earlier. It then makes a check to see if the computer is in a winning position or not. (Program lines 6000-7000. Flowchart Fig. 13).

Section 3 is where the program starts, and it initialises the values of variables and sets up the board. (Program lines 10-160. Flowchart Fig. 14).

Section 4 deals with the computers' opponents' move. (Program lines 163-180 and 2000-3030. Flowchart Fig. 15).
Section 5 in conjunction with Section 2 enables the computer to evaluate and play its moves. (Program lines 190-360. Flowchart Fig. 16).
We will now go through each of

these sections in turn in more detail.

SECTION 1

If you look at Flowchart Fig. 12, you will see that we print a heading (Pile No.   No. of matches) and then set up a FOR NEXT loop to print the pile numbers and the number of matches in the piles under this heading. The piles themselves are stored in A array locations A(1) to A(A) where 3 ≤ A ≤ 6) and notice that after each pile has been printed, it is checked to see if it is empty. If it is, then we branch round to the NEXT Q statement. You should see from this that if all ther piles are empty the variable Z will still have the value 0 that it received at the start of the routine. After all the piles have been printed, we test the value of Z and if it is 0 we set F to 1. This means that when we jump to this subroutine to print the board, a check is also made to see if the move just played has enabled one of the players to win the game. If it has, we set F to 1 to signify this fact (this technique is called setting a flag to show that an event has occurred).

SECTION 2

What this subroutine is doing (see Fig. 13.) is to say, "if the computer were to take R matches from pile P, would that leave the oppostion in a losing position." The first thing to do is to take R matches from pile P and this is done in the first box. We then set three variables to 0, (Z, U and I) which will be used later to keep a decimal total of the binary digits. The next box sets up a FOR NEXT loop in Q from 1 to A (A is the number of piles). The next step is to convert the number of matches in each pile in turn into binary digits being stored in variables V, B and M. We then add V, B, and N to Z, U and I respectively to keep a running decimal total of the binary numbers. The next box we encounter is NEXT Q, which branches us back to deal with the provision of the next pile.

When all the piles have been converted to binary and added up into Z, U and I, we put R matches back into pile P and then check each of the digits Z, U and I in turn to see if it isodd or even. If any of these digits are odd, we branch to the RETURN statement. Only if they are all even will the variable O be set to 1 to flag the fact that making R from pile P is a winning move for the computer. ▶

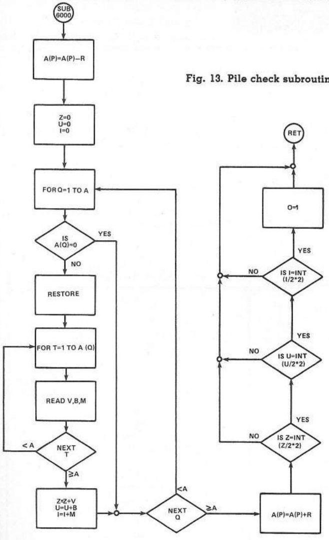

SUB 6000
A(P)=A(P)−R
Z=0
U=0
I=0
FOR Q=1 TO A
IS A(Q)=0
YES
NO
RESTORE
FOR T=1 TO A (Q)
READ V,B,M
NEXT T
< A
≥A
Z=Z+V
U=U+B
I=I+M
NEXT Q
<A
≥A
A(P)=A(P)+R
IS Z=INT (Z/2*2)
NO
YES
IS U=INT (U/2*2)
NO
YES
IS I=INT (I/2*2)
NO
YES
O=1
RET

Fig. 13. Pile check subroutine

```
DATA 0, 0,1,0,1,0,0,1,1,1,0,0,1,0,1,1,1,0,1,1,1
```

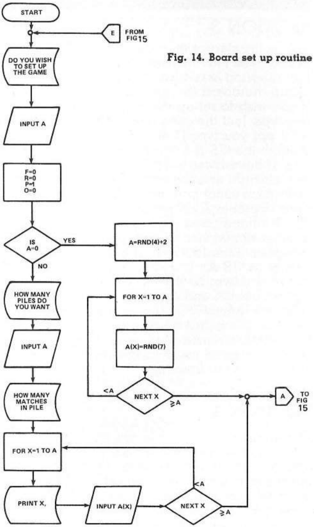
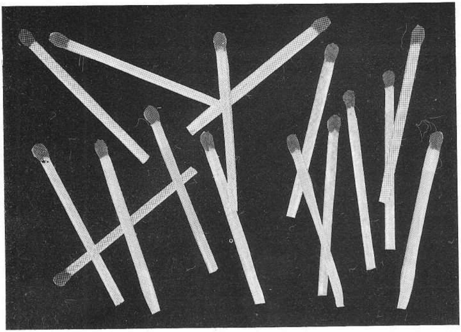

START
E
FROM FIG15
DO YOU WISH TO SET UP THE GAME
INPUT A
F=0
R=0
P=1
O=0
IS A=0
YES
NO
A=RND(4)+2
FOR X=1 TO A
A(X)=RND(7)
NEXT X
<A
≥A
A
TO FIG 15
HOW MANY PILES DO YOU WANT
INPUT A
HOW MANY MATCHES IN PILE
FOR X=1 TO A
PRINT X,
INPUT A(X)
NEXT X
<A
≥A

Fig. 14. Board set up routine



SECTION 3

This is the start of the game proper (see Fig. 14) and the first question asks who should set up the board for the game. If you wish to set up the board, you type 1; if the computer is to set it up, you type 0. We then come to the "IS A = 0" decision box. If the answer is YES, we branch right and the computer will pick a random number of piles (variable A such that 3 <A> 6) and put a random number of matches in each pile (program lines 150-160). If the answer to "IS A = 0" is NO, we continue down the flowchart and a question and answer session follows which allows you to set up the board (program lines 30-90). Which ever route we take, we end up at marker A and move on to flowchart Fig. 15.

SECTION 4

The first box in Fig. 15 jumps to subroutine at line 500 (Fig. 12) and prints out the position of the board that has just been set up in section 3. It is not expected that F will be set to 1 at this point, and so it is not checked. The next three boxes deal with the question of who should start first, and Fig. 15 shows what happens if you choose to move first. You are asked which pile you wish to take from, and how many matches you wish to take from this pile. These matches are then removed (program lines 2000-2035). We then branch to subroutine at line 500 (Fig. 12) to print the board and since a move has been made, the flag F is tested upon return. If the flag is set (which it shouldn't be after only one move) you are told you have won, and asked if you wish to play again. If you do, the program branches via marker E back to the start box in Fig. 14. If not, the program ends. Assume, however, that your first move did not enable you to win! (The reason for the check at this point is because this section is used every time you make a move and sooner or later you may well win). The program then branches to marker B in Fig. 16 which is the

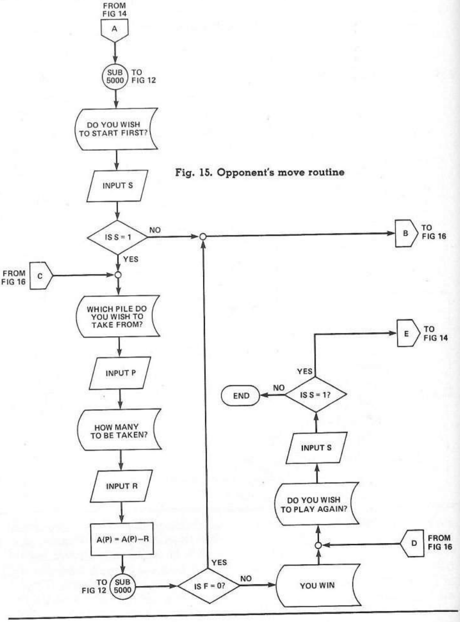

FROM FIG 14
A
SUB 5000
TO FIG 12
DO YOU WISH TO START FIRST?
INPUT S
IS S = 1
NO
B
TO FIG 16
YES
FROM FIG 16
C
WHICH PILE DO YOU WISH TO TAKE FROM?
INPUT P
HOW MANY TO BE TAKEN?
INPUT R
A(P) = A(P)−R
TO FIG 12
SUB 5000
IS F = 0?
YES
NO
YOU WIN
FROM FIG 16
D
DO YOU WISH TO PLAY AGAIN?
INPUT S
IS S = 1?
NO
END
YES
E
TO FIG 14

Fig. 15. Opponent's move routine

same place as you would have reached if you had decided to let the computer play first.

SECTION 5

This section works out the computer's move (see Fig. 16) and when you see the amount of work done, you will also see that it takes full advantage of the computer's speed. What happens in effect is that the computer starts with pile 1 (P= 1) or the first pile that contains matches (A (P) > 0) and takes 1 match from the pile (R= 1). A branch is then made to the subroutine at line 6000 (Fig. 13) where a check is made to see if this is a winning move (O= 1). If it is, then this move is made. If it is not a winning move, then another match is taken from the pile (R= 2) and the check is made again. Checking continues through all the matches (R= 1TO A[P]) in this pile and if no winning move is found, the computer moves on to the next pile that contains matches and checks all the possibilities in this second pile. This continues until either all the piles have been checked and no winning move is found, or until taking R matches from pile P produces a winning position (O= 1). If O ever equals 1, that move is made. If no winning is found, then a move is made at random. After the computer has made its move, we GOSUB 500 to print the current board and to check for a win. If the computer's move has given it the game (F= 1) this fact is printed, and we branch to marker D in Fig. 15 (Do you wish to play again?).

NIM is in fairly general BASIC and should be easily adaptable to play on most micros.

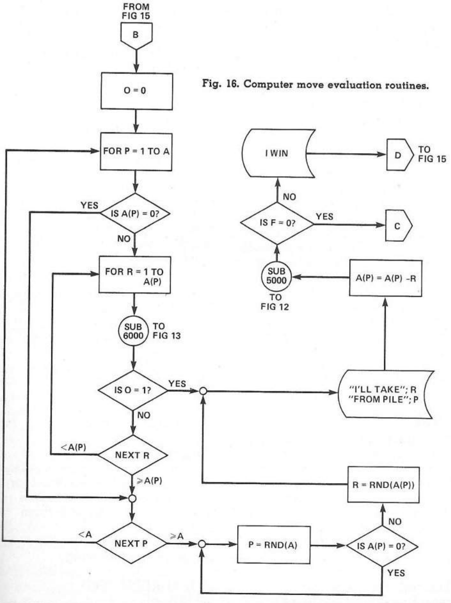


Fig. 16. Computer move evaluation routines.

AGAIN?

If the computer has not won by its move, we branch to marker D in Fig. 15 for you to make your next move. At this point, I am taking my 23rd aspirin, and gratefully declare that to be it for this time. Probably the best thing to do now would be to go through and master the above program and flowcharts.

Next time, we'll take a break and make a start on extended BASIC.

Phil Cornes

BEGINNING BASIC – 8

We investigate Extended BASIC and see what it can do.

Up to now, all the facilities we have described should be found on any machine that can run any form of BASIC. Different manufacturers tend to have different ideas on the facilities that should be provided, and they tend to pick some facilities because they highlight some of the best points of their machines.

Having said this, however, ther are still plenty of facilities common to most machines: just don't be too upset if we describe a facility your machine doesn't have, or miss a facility it does have.

When we first started the series, we took a look at the meaning of words like variable, and operator. We are going to go back and look at these again now, as their scope has been broadened somewhat with the introduction of Extended BASIC.

VARIABLES

The first thing to note about Extended BASIC is that the number of variable names increases somewhat. Whereas in Tiny BASIC we had 26 variable names, A-Z, and one single subscript variable, A(X) or @(X), in Extended BASIC we have many more. Typically these include:

1. The letters A-Z:
2. Any letter followed by a single digit 0-9 (eg A1, S5, Z9 etc. where A1 is totally distinct from A(1) and so on;
3. Many BASIC versions also include combinations of two or more letters (eg ZQ, PT, ID etc);
4. In addition to this, any common variable name (T, B4, PR etc.) may also be made into a subscripted variable of the form, T(X), B4(X), PR(X) etc.
5. String variables of the form any common variable name followed by a $ sign (eg B$, C8$, ST$ etc).

The first three types of variable name (common variables) listed above are quite straight forward. They are just an extension of Tiny BASIC's 26 variable names A-Z, but the other types of variable will require a little more discussion.

SCRIPT VARIABLES

In Tiny BASIC we had A(X) where X could take any value from 1 to a value determined by the amount of free memory space available. Since we were only allowed one such variable, we did not need to inform the computer how many elements of this array we were going to use because the computer would keep accepting the values assigned to the various elements of the array until it ran out of free memory space and informed us of this fact.

In Extended BASIC, there are endless numbers of possible Subscript variable names and the computer has to allocate a known amount of memory to each one that appears in a program. This means that we now have to tell the computer how many elements will be used for each subscript variable. We do this by means of a new statement.

DIM

No! This is not to tell the computer that we think it is thick. DIM is an abbreviation for the word DIMension where the word is used to mean size in this context. The following is an example of its use:

```
10 DIM Q(4)
```

This tells the computer to reserve memory space for 5 elements to be labelled Q(0), Q(1), Q(3) and Q(4).

It is possible to dimension several variables in a single DIM statement, as follows:

```
30 DIM B4(3), AD(7), R(19)
```

This statement is telling the computer to reserve space for 32 subscript variable elements all together (don't forget we also include zero subscripted elements (B4(0) etc.) now).

In Tiny BASIC our subscript variable only had one DIMension. Just to confuse you, one dimension here means that there was only one number in the bracket to specify which element we were referring to. In Extended BASIC, we can have subscripted variables in more than one dimension.

```
56 DIM D(3,2)
```

tells the computer to reserve memory space for the D array which has double subscripts o-3 and 0-2 (12 elements in all). You can imagine this to be a two dimensional matrix set out as follows:

D(3,0)	D(3,1)	D(3,2)
D(2,0)	D(2,1)	D(2,2)
D(1,0)	D(1,1)	D(1,2)
D(0,0)	D(0,1)	D(0,2)

Some versions of BASIC will allow more than two dimensions:

```
71 DIM GT(3,7,6,4)
```

contains four dimensions. Indeed I have seen one version of BASIC which will allow 9 or 10 dimensions (heaven knows what you would do with them all!).

Consider the following:

```
10 DIM A(3,2), V(2)
20 V(0)=0
```

```
30 V(1)=8
40 V(2)=12.5
50 FOR X=0 TO 3
60 READ PR
70 FOR Y=0 TO 2
80 A(X,Y)=PR+PR*V(Y)/100
90 NEXT Y
100 NEXT X
110 DATA 530,630,704,931
120 END
```

This program is calculating VAT at the three different rates, 0%, 8% and 12.5%, on the prices of items listed in the DATA statement of line 100. The three VAT rates are stored in the V array, [V(0) to V(2)] by lines 20 to 40.

Each price in line 110 then has VAT added to it by line 80. All the answers are stored in the A array, a two dimensional array. Each of the columns in this array stores the three prices, one for each VAT rate, and each row stores the price of the four items at a single VAT rate.

Incidentally you now know enough BASIC at this point to write programs to perform the three card shuffling routines that we started with. If you want to have a go you will find some suggested solutions in Fig. 7.

STRING VARIABLES

This is a totally new kind of variable and one that we shall spend much time discussing as there are many facilities associated with it.

Just as a common numeric variable (A,X,Z etc) can be assigned a value which can then be manipulated and used in calculations and decisions, so can a string variable be assigned a value which can be used similarly. The main difference is that a string of characters that are available on the keyboard (usually with three exceptions — comma, inverted commas and carriage returns). Eg:

```
10 A$= "THIS IS THE STRING
CALLED A$"
```

In this example, the computer will assign to the variable A$ the value

```
THIS IS THE STRING CALLED
A$
```

note that the inverted commas are not assigned to A$. They are used by the computer to show where the string begins and ends.

As with any other variable, the statement

```
30 PRINT A$
```

would cause the above message (minus inverted commas) to appear on the output peripheral.
Consider the following —

```
5 PRINT "INPUT YOUR NAME"
10 INPUT A$
20 IF A$ = "PHIL" THEN 50
30 PRINT A$; "IS NOT
   ACCEPTABLE —
   PROGRAM ENDED"
40 END
50 PRINT "HI PHIL — WHAT'S
   ON TODAY"
60 ........
70 ........
```

Here we see two more examples of string variables being used in the same way as numberic variables can be used.

Line 5 asks you to INPUT YOUR NAME. Line 10 will assign any string of characters you input to A$. You do not need to input string starts and ends. Line 20 checks your input string for a particular combination of characters (in this case PHIL) and if this combination is found, the program branches to line 50 and continues.

If your input string is not the particular combination being considered then the program terminates in line 40 after printing an error message in line 30.

It would have been quite acceptable to use any relational operator (= >, <, >, < = ,< >) instead of the = sign in line 20. For example, suppose we had used =, what would this mean? In ASCII code (the most popular computer code) every character is given a 7 bit binary number, as its representation, so that —

A in ASCII is 1000001
B is 1000010
C is 1000011 etc

in ascending binary order, so when the computer is faced with

```
20 IF A$   = "PHIL" THEN 50
```

then it will compare the first character of the word PHIL (P= 1010000 ASCII) with the first character of A$. If the ASCII for the first character of A$ is less than 1010000 then the test fails. If the ASCII for the first character of A$ is greater than 1010000 the test succeeds. If the two ASCII codes are equal, then the computer knows that the two words have teh same first letter. It does not know the relationship between subsequent letters, and so these have to be checked — second letter of A$ with second letter of "PHIL" etc — until the test fails with one of the letters of A$ being less than one of the letters of PHIL, or passes with one of the letters of A$ being greater than one of the letters of PHIL, or passes with all of the letters of A$ being the same as all the letters of PHIL. Therefore, if —

A$ = "PHI" the test will fail (A$<PHIL)
A$ = "PHIL" the test will pass (A$ = PHIL)
A$ = "PHILIP"the test will pass (AS>PHIL)

PH<PHIL because the letter L in PHIL will be compared with the fourth letter of PHI which is a NUL character, which has ASCII code 0000000 and is therefore the least of the ASCII codes (the same reason applies as to why PHIL<PHILIP).

We will now make a start on some of the string functions available in Extended BASIC.

MID$ (STRING,S,L)

It would be most useful if it were possible to extract characters at will from within a string so that they could be tested or manipulated separately (we will see an example of this later) and, indeed, it can be done using the MID $ string function.
Consider the following —

▶

```
10 A$ = "STRING"
20 B$ = MID$(A$,3,4,)
30 PRINTS B$
40 END
```

the output from this program would be the word RING.

The MID$ function tells the computer to return a substring of the specified STRING variable (here A$) starting at position S (here 3) and containing L characters (here 4).

The word STRING in the heading above may be replaced with any string variable name or string expression, and the variables S and L may be any numeric constant, variable name or numeric expression.

The following is a short program which reads a string of characters from a data statement and searches through it to find the start position of a three-character sub string which is also contained in the DATA statement.

```
10 READ A$, BS
20 L = 3
30 S = 1
40 T$ = MID$(A$,S,L)
50 IF T$ = B$ THEN 80
60 S = S+1
70 GOTO 40
80 PRINT B$; "STARTS AT
   POSITION";S;"OF";A$
90 DATA "EDUCATION",
   "CAT"
100 END
```

If this program were run, its output would be

```
CAT STARTS AT POSITION 4
OF EDUCATION
```

Just for practice, look through this program and see if you can see how it works.

Before we finish for this time, we will look at just one more of the string functions available to Extended BASIC because you will need it for this time's homework.

LEN (STRING)

The LEN function returns a numeric value equal to the length of the string in the brackets, so that —

```
10 A$ = "PHIL"
20 L = LEN(A$)
```

would assign a value of 4 to L. Similarly, we could have said —

```
20 L = LEN ("PHIL")
```

and L would have taken the same value.

There is an old saying which says that you should only eat pork in months whose names contains the letter R. So you could eat PORK in MaRch or SeptembeR, but not in May or June.

As an exercise try to write a program which will ask for name of a month to be input, accept an answer as a string, and then search through the input, character by character, looking for an R. If an R is found, a message telling you that you may eat pork in this month should be printed. If no R is found, the opposite message should be output. So if the input was APRIL, the output would be, YOU CAN EAT PORK IN APRIL etc.

Test your program to make sure that it works by using the following test input data.

1) MAY
2) OCTOBER
3) MARCH
4) ENGLAND

Then we shall go on to examine some more string functions.

Phil Cornes

BEGINNING BASIC – 9

We look at more aspects of Extended BASIC.

The problem at the end of the last article may may have increased the aspirin intake of some of you. The problem required that you write a problem which will ask for the name of the month to be input, accept an answer as a string and then search through the input string for the occurence of the letter R.

SEARCHING STRINGS

If an R is found, the program should say that YOU CAN EAT PORK IN your input month. If no R is found, the opposite message shall be output. The problem in effect boils down to checking through an input string to find a particular character. If the character is present, an output to this effect is given; otherwise the opposite output is printed.

Consider the following:

```
40 PRINT "INPUT MONTH"
50 INPUT M$
130 L=LEN (M$)
140 FOR N= 1TO L
150 Q$= MID $ (M$,N, 1)
160 IF Q$= "R" THEN 200
170 NEXT N
180 PRINT "YOU CANNOT
    EAT PORK IN";M$
190 END
200 PRINT "YOU CAN EAT
    PORK IN",M$
210 END
```

Here you are asked to input the name of a month (lines 40 and 50) then a check is made to find the length of the input string (M$) (line 30) which is assigned to variable L.

We now need to make a check character by character to test for the chosen character (R here). To do this we set up for a FOR NEXT loop in N between 1 and L. We then extract the Nth character of M$ (line 150) and assign this to Q$. This character is then checked to see if it is an R. If it is, we branch ti line 200 and PRINT:

"YOU CAN EAT PORK IN";M$

If the character is not an R we branch back to check the next character of M$ (line 170).

At first sight this program appears to work quite well, but if we look at the test data specified last month then we run up against a problem. The test data was:

1) MAY
2) OCTOBER
3) MARCH
4) ENGLAND

If these were entered we would get the following output:

1) YOU CANNOT EAT PORK IN MAY
2) YOU CAN EAT PORK IN OCTOBER
3) YOU CAN EAT PORK IN MARCH
4) YOU CANNOT EAT PORK IN ENGLAND

USING DATA STRINGS

The problem as set doesn't require the computer to pass judgement upon the location, but upon the month. Unfortunately, we have not given the computer any method of "knowing" whether the input string is a month or any other word, the computer just treats it as a string. If we want only months to be dealt with, then we have to tell the computer how to check this out.

One solution is to give the computer a list of the 12 acceptable inputs in a DATA statement and then check the input M$ against each of these to seek a match before going on to the rest of the program. This is done in the program below by lines 60-120 and 300-320. Apart from these additions, the program is the same as the previous one.

```
40 PRINT "INPUT MONTH"
50 INPUT M$
60 RESTORE
70 FOR C=1 TO 12
80 READ T$
90 IF T$= M$ THEN 130
100 NEXT C
110 PRINT M$;"IS NOT A
    MONTH NAME"
120 END
130 L=LEN(M$)
140 FOR N= 1 TO L
150 Q$= MID$(M$,N, 1)
160 IF Q$= "R" THEN 120
170 NEXT N
180 PRINT "YOU CANNOT
    EAT PORK IN";M$
190 END
200 PRINT "YOU CAN EAT
    PORK IN";M$
210 END
300 DATA JANUARY,
    FEBRUARY, MARCH,
    APRIL
310 DATA MAY, JUNE, JULY
    AUGUST, SEPTEMBER
320 DATA OCTOBER,
    NOVEMBER, DECEMBER
```

I leave you to make it work. Carrying on now, we will go on to look at some more string functions.

LEFT $(STRING,L)

LEFT $ is similar to MID$ in that it allows you to extract a substring from the string specified in the brackets, but here the substring will consist of the first L characters of STRING eg

```
10 A$= "TIMOTHY"
20 B$= LEFT$(A$,3)
30 PRINT B$
```

The output from this program segment would be TIM which is a substring (first 3 characters here) of the string TIMOTHY.

RIGHT S(STRING,L)

This function works the same as the previous one except that it extracts the last L characters

▶

from the string in the brackets, eg.

```
10 B$= RIGHT$("BULLOCK"
,4)
20 PRINT B$
```

Here the output would be LOCK.

STR$(X)

This function converts the numeric value of the contents of the brackets into a string, eg.

```
10 A= 126
20 A$= STR$(A + 12)
30 PRINT A$
```

Here the PRINT statement would output a value of 138 for A$. (This should not be confused with a value 138 assigned to a numeric variable — the 1, 3 and 8 in A$ are treated purely as characters.)

The contents of the brackets in a STR$ statement can be any constant, variable or numeric expression.

VAL (STRING)

This function has the opposite effect to the previous function. It converts a string with a numeric value back into numbers which can be assigned to a numeric variable, eg

```
10 A$= "13"
20 B$= "15"
30 C= VAL(A$+ "." + B$)
40 PRINT C
```

Here the value of C printed will be 13.15 because the + sign in a string expression does not mean add in the normal arithmetic sense, it means that the various strings should be lined up after each other in the order given (this is called concatanation). Similarly, we could have added a line 25 to the above segment as follows:

```
25C$= A$+ "." + B$
```

This would have been quite acceptable and C$ would have taken the value of 13.15 but this time it would have been a string not a number.

CHR$(X)

This function returns the single character string which has X as its ASCII code. The value of X in this function if in decimal and has a range 0-127 (as there are 128 ASCII codes). The variable X in the brackets may be replaced by any constant, variable or numberic expression. For example,

```
10 FOR X = 65 TO 90
20 PRINT CHR$(X);
30 NEXT X
```

This would print out all the letters of the alphabet in order from A to Z as the numbers 65-90 are the decimal equivalent of the ASCII codes for these letters. Similarly, the numbers 0-9 have codes 48-57 as the decimal of the ASCII codes.

ASC (STRING)

This has the opposite effect of CHR$ in that it takes the first character (first character only) of the string in the brackets and returns the decimal equivalent of its ASCII code as a number to be assigned to a numeric variable. Eg,

```
10 A$= "STRING"
20 A = ASC(A$)
30 PRINT A
```

Here A takes the value 83, which will be printed by line 30, as this is the decimal equivalent of the ASCII code for the letter S.

That completes a rather impressive list of string functions. There are one or two others which can be found on some machines, but they are not really standard and will not be introduced.

NUMERIC FUNCTIONS

We will now go on to examine the other big plus of Extended BASIC — Extra numeric functions. In all the examples below, the contents of the brackets can be replaced with either a constant, variable or numeric expression.

We have already covered 4 functions in Tiny BASIC; these were ABS(X), INT(X), TAB(X), and RND(X). These four functions are still available and join the rest of the list presented below.

LOG(X)

This function returns the log to base e of the contents of the brackets, eg:

```
10 A = LOG(3)
```

here A takes the value 1.09861.

EXP(X)

This function raises the base e to the power X and is thus the inverse of the LOG(X) function.

```
10A = EXP(3)
```

here A takes the value 20.0855

SGN(X)

This is the Signum function. If the contents of the brackets have a -ve sign then this function returns a value – 1; if the contents are + ve the function returns + 1; and if the contents are zero, the function returns zero.

```
10A = SGN( – 12)
```

here A takes the value – 1.

SQR(X)

This function returns a value to the square root of the contents of the brackets (obviously these contents must not be negative).

```
10 A = SQR(9)
```

here A takes the value 3.

TAN(X)

This function returns the trigonometric Tangent of the contents of the brackets (most trig functions assume X to be in radians – 1 degree = 0.01745 radians and 1 radian = 57.2957 degrees).

SIN(X) AND COS(X)

These functions return the trig Sine and Cosine respectively (again X assumed to be in radians).

ATN(X)

This function returns the Arctan (the angle in radians whose Tangent is X) of the contents of the brackets.

This ends the arithmetic functions: in addition most machines have two special functions, PEEK (X) and POKE L,C.

PEEK(X)

This function returns a decimal integer in the range 0-255 corresponding to the decimal value of the binary code contained in the computer memory location X. X is also in decimal and can be any constant variable or numeric expression.

POKE L,C

This function is the inverse of the PEEK function. It allows you to enter any 8 bit binary combination into any location of RAM where L is the location number and C is the code to be entered (both L and C are in decimal).

In this section there has been quite a lot of new material introduced and you would be well advised to re-read it if you are not familiar with it.

There are two good exercises you could now try:

1. Write a short program which will accept a number from 0-255 as input and convert this into a two digit Hexadecimal number on the range 00-FF.
2. Write a short program which will do the reverse of the above — accept a 2-digit Hexadecimal number and convert it to decimal.

And for all you bright boys (or those who don't know about Hexadecimal) you might also like to look at the problem of how you would go about taking a list of numbers in random order and sorting them into numerical order as we shall be moving on to such things from the next article.

EPSON
HX-20
DRAGON 32

Dragon 32 and the Epson HX-20 use Extended BASIC.

Phil Cornes

BEGINNING BASIC — 10

We continue our look at Extended BASIC functions.

The first two of the last article's problems to make sure you get the hang of all the string functions and Extended BASIC generally. The first problem was to convert a decimal number in the range 0-255 into 2 digits HEX.

Hexadecimal is a number system based on 16 different "digits". These range from 0-9 and for the other 6 we use letters A to F. In Hexadecimal, then, we count 00, 01, 02......0F, 10, 11......1F, 20, 21......2F, and so on up to FF. In each digit position we can put any one of 16 different symbols so that in a 2 digit Hex number there are 16 x 16 (ie 256) different permutations of symbols. This means that if we start from zero we can respresent any integer from 0 to 255 with 2 hex digits. Conversely, any integer from 0 to 255 needs only 2 Hex digits to represent it. Before we delve into the two Hex digit conversion, let us look at a method of converting an integer 0-15 into a single Hex.

HANDLING HEX

The first point to note is that our single Hex digit can either be a number 0-9 or a letter A-F. This means that we cannot use a numeric variable to store the Hex digit (we cannot store character A-F in a numeric variable) so we will have to use a string (say A$).

If you remember from the last article we looked at the CGR$(X) function. This function returns a single character string which has X (in decimal) as its ASCII code. If we decide to utilize this function (which we will) to convert from ASCII code to a Hex character then all that is required is to convert the input number 0-15 into the ASCII representation of its HEX equivalent.

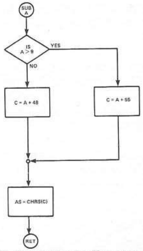


Fig.17. Subroutine for Hex conversion.

Consider Fig. 17:
Let A be the number 0-15 that we wish to convert. You may remember that the ASCII codes for the digits 0-9 are 48-57 and the ASCII codes for the letters A-F are 65-70.

This means that if the number we wish to convert is between 0-9 we should add 48 to A to give the ASCII code of the Hex digit and if A is between 10-15 we should add 55 to give an ASCII code between A-F. The resulting value of C (ASCII code) can now be converted by the CHR$(X) function to the Hex character representation of the input value of A.

We can extend this idea to produce two Hex digits. The most significant Hex digit in a two Hex digit number is the number of 16s in the decimal number and the least significant Hex digit is the remainder. For example 250 in decimal is FA in Hex because there is 15 x 16 in 250 and 10 remainder. 15 in Hex is F and 10 in Hex is A.

AN EASY SOLUTION

To solve our problem therefore what we need to do is to divide the input number up into two decimal numbers between 0-15 each of which can then be converted separately into Hex digits by the subroutine of Fig. 17. This procedure is laid out in Fig. 18 (to save re-drawing it, Fig. 17 is called as a subroutine of Fig. 18). Both of these flow charts can be coded into BASIC as follows.

```
 10 INPUT N
 20 A=INT(N/16)
 30 GOSUB 500
 40 B$=A$
 50 A=N—A*16
 60 GOSUB 500
 70 B$=B$+A$
 80 PRINT B$
 90 END
500 IF A     9 THEN 530
510 C=A+48
520 GOTO 540
530 C=A+55
540 A$=CHR$(C)
550 RETURN
600 END
```

NOTE: The END statement of line 90 is needed otherwise the computer will "crash" into the subroutine at line 500 straight after the printout of B$ by line 80 even though there is no subroutine call. Thus when the computer reaches line 550 it will execute a return statement and find that there is no return address stored and "bomb out".

Similar principles to those given above using the ASC(STRING) function can be applied to the second question of the homework. The flow charts of Figs. 3, 19 and 20 and the program below are given as a possible solution without explanation. It is left to you to sort out how they work (good practice).

```
10 INPUT A$
20 B$=LEFT$(A$,1)
30 GOSUB 500
40 N=A*16
50 B$=RIGHT$(A$,1)
60 GOSUB 500
```

```
 70 T=T+A
 80 PRINT T
 90 END
500 C=ASC(B$)
510 IF C  57 THEN 540
520 A=C-48
530 GOTO 550
540 A=C-55
550 RETURN
600 END
```

Right! Having solved the first two problems we will now go on to look at the third thing we mentioned — sorting. Approximately 95% of all computer time used anywhere is used by commercial concerns in the process of running a business. Very little computer time is used for scientific applications. Of this vast amount of computer time used commercially, about one third is spent sorting and searching through lists of data items so that useful information may be extracted. It is to these two areas, sorting and searching that our attention now turns and now we will look at a simple search and at how to merge two sorted lists together.

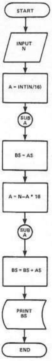


Fig.18. Complete BASIC flowchart for conversion program.

SIMPLE SEARCH

Imagine a list of names, addresses amd telephone numbers. If we are given a name we can look through the list for this name and when we find it we can look up the corresponding address and telephone number. This process is called searching and the item we are using to find the required information is called key. Here we are using the name to find the address and telephone number so the name is the key. If the item in the list are in random order, then there is really only one way of finding any given item; that it to look through the list from the beginning, one item at a time, until the required item is located.

So the flowchart of an algorithm to perform this simple searching technique might be as Fig. 21. The following notes might help you to understand Fig. 21 better. N is the number of items in the list and K is the number of the item that we are currently looking at. The list itself might well be contained in a string array, eg A$(0) to A$(9) is a list of 10 elements each of which would be capable of storing 256 characters.

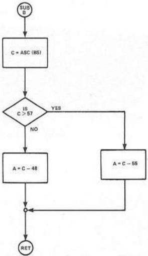


Fig.19. Subroutine for Hex to Decimal conversion.

Working through Fig. 21 the first box asks that we input the key that we are to search for (eg input the name whose address and telephone number we seek). We then set up for a FOR NEXT loop from 1 to the number of items in the list. The third box extracts the key from the Kth item of the list. This is necessary because it is quite possible that the search key might not be at the beginning of each item. For example, if our list were in telephone number order with telephone numbers specified first, then address specified second, and lastly name, we would need to extract the last so many characters to compare them with our input. This comparison is carried out in the 4th flowchart box. If it turns out that the extracted key from the Kth item on the list is the same as the input then the information (address and telephone number) is printed out. If the two items are not the same, K is incremented by one and the key is extracted from the next item in the list to be checked. This process is continued until the extracted key matches the input or until the computer has checked through all the items on the list and not found a match in which case a message to this effect is printed.

TWO LIST MERGING

Imagine two lists of numbers that have already been sorted out into numerical order and imagine further that these two lists have been assigned to two array variables, eg

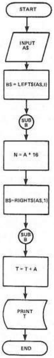


Fig.20. Complete BASIC flowchart for reverse conversion.

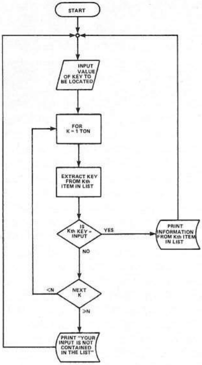
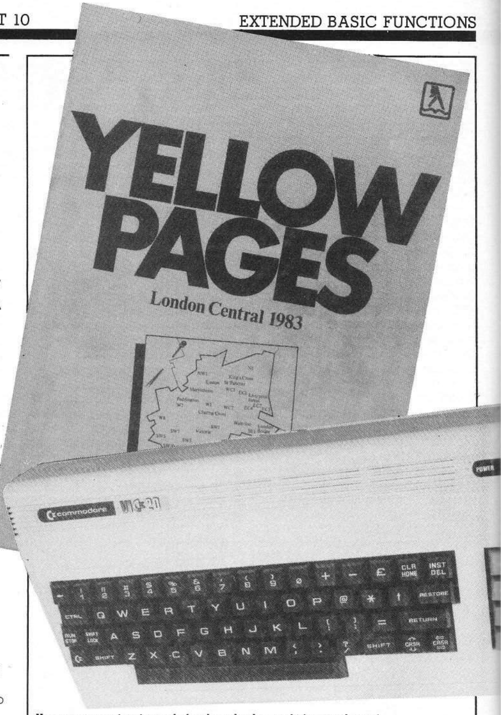

START
INPUT VALUE OF KEY TO BE LOCATED
FOR K = 1 TON
EXTRACT KEY FROM Kth ITEM IN LIST
IS Kth KEY = INPUT
YES
NO
PRINT INFORMATION FROM Kth ITEM IN LIST
NEXT K
<N
>N
PRINT "YOUR INPUT IS NOT CONTAINED IN THE LIST"

Fig.21. Search routine flowchart.

	A(X)	B(X)
X	A(X)	(X)
1	1	−1
2	3	2
3	12	13
4	24	15


YELLOW PAGES
London Central 1983
commodore VIC-20

You can use a micro instead of a phone book — and it's not as heavy!

It is required that the two lists A(X) and B(X) be merged together to produce a single list containing 8 items and that the merged list say C(X) should also be in ascending numerical order.

The method of solution for this problem is to compare the first item in array A with the first item in array B. The least of these should be made the first item of array C. In this case B(1) is less tham A(1) so C(1) will be set equal to B(1) (ie −1). We will now compare A(1) with B(2), the least of these will become C(2). Here A(1) B(2) so C(2) = A(1) = 1 and we then compare A(2) with B(2) and put the least of the these into C(3) etc. A program to perform this algorithm is given below without any further explanation so that you can dry run it yourselves.

```
 10 DIM A(4),B(4),C(8)
 15 FOR X= 1TO4
 20 INPUT A(X),B(X)
 25 NEX X
 30 A= 1
 35 B= 1
 40 C= 1
 50 IF A(A)  B(B) THEN 90
 60 C(C)=A(A)
 70 A=A+1
 80 GOTO 110
 90 C(C)=B(B)
100 B=B+1
110 C=C+1
120 IF A=5 THEN 150
130 IF B=5 THEN 190
140 GOTO 50
150 FOR D=BTO4
160 C(D+4)=B(D)
170 NEXT D
180 GOTO 220
190 FOR D=A TO 4
200 C(D+4)=A(D)
210 NEXT D
220 FOR C= 1TO8
230 PRINT C(C)
240 NEXT C
250 END
```

For practice try to work out what lines 120-210 of the two list merge program below are included for and when you have worked it out try to think of another way of doing this which although not as general will reduce the length of the program listing considerably.

Phil Cornes

BEGINNING BASIC – 11

In this article we present the solution to the sort problem and take a look at Binary searches.

The object of the problem in the previous article was to try to decide what lines 120-210 of the two list merge program are included for. You remember that the program was to take two lists of numbers, A(X) and B(X), each containing 4 items in ascending numerical order and produce from these a single list, C(X), of 8 items which was to have been the result of merging the two lists together in such a way that the resulting list was also in ascending order. A simple algorithm was given to solve the problem and you were expected to dry run the program listing produced from this algorithm. The best way to try to understand the operation of a program like this is to draw a near flowchart. (See Fig.22).

As far as possible, the flowchart boxes have been numbered to correspond with the relevant program line numbers given. The shaded area in Fig.22 corresponds to lines 120-210 of the program. The flowchart boxes labelled 15, 20 & 25 allow you to input the two number lists A(X) and B(X). Note that these must be in ascending numerical order. Flowchart box 35 sets up initial values for A, B and C which will be used as pointers in lists A(X), B(X) and C(X) respectively. Box 50 now compares the next two items in lists A(X) and B(X) using pointers A and B to see which of A(A) and B(B) is larger (to start off with, both A and B have the value 1 so we are comparing A(1) with B(1). IF A(A) is larger than B(B) then we branch to box 90 and put the contents of B(B) into C(C). So here we make C(1) equal to B(1) (because B(1) was the smaller of A(1) or B(1) and then we increment both pointers B and C by 1. A and B are now tested to see if either of them has reached 5. In this case they have not and so we branch back to box 50 to compare A(A) with the new value of B(B). (Here A(1) and B(2)). We will suppose this time that A(1) is smaller than B(2) (though, of course, this need not be the case) so we branch to box 60 and put A(A) into C(C) (Here A(1) into C(2)). We then execute boxes 70 and 110 thus incrementing A and C by one (so now A = 2, B = 2, C = 3). A and B are now both tested again to see if either has reached 5 and as they have not, we branch back to box 50 to compare A(2) and B(2) etc. You should see that this process is repeated until one of the lists A(X) or B(X) has been exhausted so that either A or B has the value 5. When this occurs, we encounter the program lines 120-210 mentioned last month (shaded area of flowchart Fig. 22) and the process changes slightly. In box 20, originally, we only entered 4 items in each list so that when A or B equals 5 (say A) we will try to compare the next item in list B with the non-existent value A(5) unless something happens to change the flow of the program. This is in fact done. When A = 5 any items that are left in list B must all be larger than any of the items that were in list A and they must also be in ascending numerical order so that all that is requires to complete list C is to transfer the remaining items in list B directly onto the end of list C without any comparisons being necessary. This is done in boxes 150-170 (If B = 5 and A 5 then the same argument applies as before and the transfer of the last items of list A to the end of list C is dealt with by flowchart boxes 190-210).

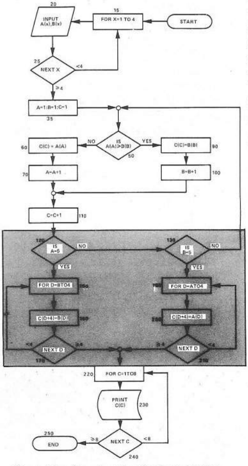


Fig.1. The flowchart for the last articles problem, the shaded area represents lines 120 to 210.

The last few flowchart boxes form a FOR NEXT loop which is used to print out the merges list C.

A SIMPLER SOLUTION

The last part of the question asked you to see if you could think of a way of simplifying the program by replacing lines 120-210 with something else. The program listing given below is one possible solution — I leave you to work out how it operates.

```
10 DIM A(5), B(5), C(8)
15 FOR X= 1TO4
20 INPUT A(X), B(X)
25 NEXT X
30 A= 1
35 B= 1
40 C= 1
43 A(5)= 1E30
47 B(5)= 1E30
```

▶

```
 50 IF A(A)>B(B) THEN 90
 60 C(C)=A(A)
 70 A=A+1
 80 GOTO 110
 90 C(C)=B(B)
100 B=B+1
110 C=C+1
120 IF C=9 THEN 220
140 GOTO 50
220 FOR C=1TO8
230 PRINT C(C)
240 NEXT C
250 END
```

NOTE: For those of you who have not met scientific representations of numbers before, 1E30 in lines 43 and 47 means 1 x 10^{30} or 1 with 30 zeros after it like this: 1000000000000000000000000-0000000. As you can see; it's quite a large number and much easier to write as 1E30.

SIMPLE SORT

So far we have examined a simple search algorithm and a merge algorithm, before we go on to consider anything more complex in either of these directions, we will take a look at a simple sorting algorithm and program.

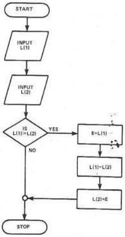


Fig.23. Simple sort algorithm for increasing order of 2 items.

Imagine a list containing two items L(1) and L(2) and now imagine an algorithm to place these two items in ascending numerical order — easy! I hear you say, and so it is. (See Fig.23.) But now imagine this list increased to contain 3 items L(1)—L(3). Can we still sort them? Yes, obviously we can and in very similar fashion (see Fig.24).

You should begin to see a general algorithm appearing from all this which can be used for any length of list (N). For P=1 to N, compare L(P) with L(P+1) and reverse these if necessary, then go on to the next value of P. This FOR NEXT loop should be repeated until we manage to go once completely through the FOR NEXT loop without having to reverse any pair of numbers L(P) and L(P+1). See Fig.25 and the following program.

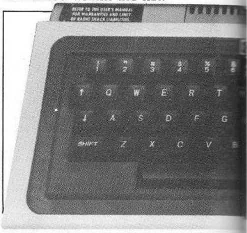
```
 10 INPUT N
 20 FOR P=1 TO N
 30 INPUT  L(P)
 40 NEXT P
 50 F=0
 60 FOR P=1 TO N-1
 70 IF L(P)>L(P+1)THEN 110
 80 NEXT P
 90 IF F=1 THEN 50
100 END
110 F=1
120 E=L(P)
130 L(P)=L(P+1)
140 L(P+1)=E
150 GOTO 80
160 END
```

I leave you to work out the details of the operation of this program. One unusual point you will notice is that part of the work of a FOR NEXT loop in this program is carried on in a part of the program that is physically outside the loop. This is perfectly acceptable even though we have not met it before.

THE BINARY SEARCH

Now we have a sort algorithm of sorts (ouch!) there is a far more efficient search algorithm that we can use that depends for its operation upon the fact that the list of items we are to search are in numerical order. This is called "The Binary Search" for reasons that will become more obvious as we describe it. If we take any list of numbers in any numerical order and try to find one particular number on the list, then if we divide the list into two equal halves, we can say for sure that if the number exists at all in the list it must reside in one of the two halves, though we would not know which. This may seem very obvious, but if we now specify that the list is numerically ordered, then we say which half of the list contains our chosen number simply by comparing the contents of the halfway location with our chosen numbers. If these contents are larger than our number then our number must be contained in the lower half of the list and vice versa. Having eliminated half the list, we can now consider the remaining half as a complete list and split this into two and test again, and so on until we either find the item we are searching for, or prove it not to be contained in the list. This may sound a little complex when compared with the previous search algorithm but when we look at how much more efficient this algorithm is than the previous one, we will certainly see that for any reasonable length list the extra complexity is more than justified.

Before we can compare the efficiency of the two algorithms we must find a criterion that we can reasonably consider as a measure of efficiency for a search algorithm. Such a criterion may, for example, be measured as follows: Every time we look at one of the locations in the list and compare it's contents with our chosen number we call this operation one comparison. Our criterion will then be the number of comparisons we have to make before we find our chosen number or prove it is not contained within the list.



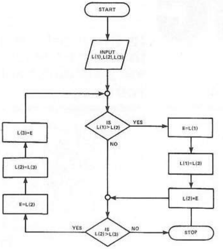


Fig.24. Sorting with three items.

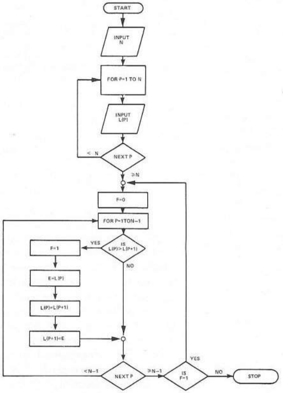
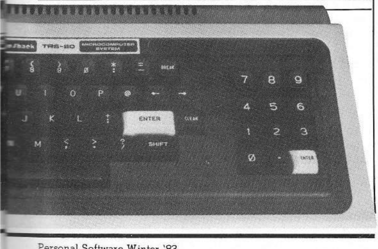


Fig.25. Flowchart for the program to perform



SPECTRUM SOFTWARE

TEST MATCH — for the 48K Spectrum **only £5.95**
This is the no. 1 hit of the summer and is a 3D Test Match Cricket simulation. This cassette contains 2 programs — the 1st a full 5-day match and the 2nd a selection of one day tests, full scoreboard and definable teams. The game already contains England & Australian teams and uses the graphics capabilities of the Spectrum to the full.

ALIEN MAZE — for the 48K Spectrum **only £5.95**
Against the clock you must decode the alien riddle that will defuse the earth shattering bomb in the second 3D Maze. At last there's a programme where the 3 dimensional graphics are not the point of the game. Keeping your head and remembering the code and where you are is a challenge for the finest mind.

GALACTIC PATROL — Spectrum and 16K ZX81 **only £5.95**
A fast machine code, arcade style, Star Trek programme with phases, torpedoes, star bases, shields and 4-types of aliens, meteors, damage control and repair and vector flight. Stunning graphics are enabled by superb machine code and there are versions for both machines on tape.

GOLF — for any 48K Spectrum **only £5.95**
Amazing 3D graphics on a memory mapped course, this programme has over 250 user definable graphics to produce a startlingly realistic simulation. You have a selection of 15 clubs and a caddy with a special blow-up of a green. The graphics have to be seen to be believed. You'll wish you had a swing as good as the cartoon golfers. There's even a 19th hole.

DERBY DAY — for the 48K Spectrum **only £5.95**
Gambling on any horse in the field, up to 5 players can lay bets with Honest Clive Spectrum the bookmaker. Will Clive keep that smile? Watch the race begin as the tape lifts and marvel at the amazingly realistic 3D animation as the riders jockey for position. See the horses and riders in full flight as they pass Spectators and into the home straight past the stands. Hold your breath at the slow motion finish. Sound and colour is used to its fullest in this 44K of superb programming. Not recommended for compulsive gamblers.

RESCUE — for the 48K Spectrum **only £5.95**
How can we summarise in a short ad, an adventure game that needs a Special Program to detail its Rules! VERY simply, you must find the Map and Radio Men plot your route and monitor patrols as they scour the 40+ locations you are travelling through. If you have the right equipment you can cross into Secret Territory in search of the Castle and the imprisoned Princess. If you find it and gain entrance there are many trails and rests. If you find the Princess you must still return to base with her. Utilises all the Spectrums facilities and takes hours to play.

JACKPOT — for the 48K Spectrum **only £4.95**
A complete simulation of a popular fruit machine, using definable graphics to the fullest. It contains a complete introduction to the rules of its HOLD, NUDGE, GAMBLE and FEATURE BOX with animated demo. Memory mapped reels, simultaneous revolution, staggered stop, animated bet and payout, payout board and realistic sound effects recreate the original. A must and a wallet saver for any fruit machine buff.

All prices include VAT & P&P **Dealer Enquiries Welcome**

Selected lines available at John Menzies, W.H. Smith & Boots.

PLEASE SEND ME THE GAMES AS TICKED:

TEST MATCH ☐ **GOLF** ☐ **ALIEN MAZE** ☐
DERBY DAY ☐ **GALACTIC PATROL** ☐ **RESCUE** ☐
JACKPOT ☐

NAME ....
ADDRESS ....
....
....
TEL NO: ....

C.R.L. **Dept PS, 140 Whitechapel Road, London E1. Tel: 01-247 9004**

PLEASE MAKE CHEQUES/PO PAYABLE TO:
COMPUTER RENTALS LTD.,

Phil Cornes

BEGINNING BASIC — 12

In the final article of this series we look at a very efficient sort routine.

We started this series with algorithms and flow charts, and that is how we finish. In this, the last part, we look at a flow chart and program for the binary search algorithm presented last time, and we also take a look at a very efficient sort routine.

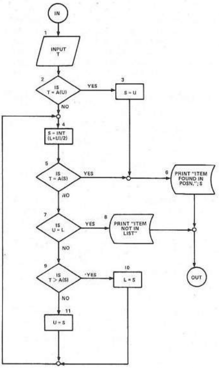


Fig.26. The Binary search flowchart.

BINARY SEARCH

A flow chart to perform the Binary Search algorithm might be as Fig.26. We assume that this routine is being used as part of a larger program so that U and L already have values. U is the pointer for the upper limit of the interval, yet to be searched and L is the pointer for the lower limit on this interval. So, for example, if the list to be searched contains 100 items, then U would nbe 100 and L would be 1. Flow chart box 1 asks for a value which will be assigned to the variable T. This is the number that we are going to search for in the list. Flow chart box 2 is looking top see iof our input value is contained in the last position of the list. If it is, we branch through box 3 to box 6 to print the message saying that we have found teh required item and to give its position (we will look at why this box is needed later).

If T is not in A(U) then we move on to box 4. This starts the algorithm proper by calculating the mid-position in the list and assigning this value to the variable S. A check is then made by box 5 to see if A(S) — the contents of the list position just calculated — is equal to T. If it is, we move to box 6 to indicate our success in location item T, otherwise we move on to box 7. Box 7 id asking whether the upper and lower list pointers are equal because if they are then there is no point in trying to locate T anymore as there are no more positions between U and L to look in, so we would move to box 8 and indicate that T was not contained in the list. If U and L are not equal, the next task is to decide if T is contained in the interval between U and S or between L and S. As the list is in numerical order, this is achieved simply by comparing T with the current value of A(S). If T is greater that A(S) then T is contained in the lower half of the list and we bring the upper bound U down to S (flowchart box 11). We than branch back to box 4 to calculate the midposition of the new interval (S) and start over again.

This process continues until we either find the item T somewhere in the list and print this fact or prove that the item T is not contained in the list at all. A program segment with the function of the flowchart of Fig.26 is given below.

```
100 .....
110 INPUT T
120 IF T=A(U) THEN 230
130 S=INT((L+U)/2)
140 IF T=A(S) THEN 240
150 IF U=L THEN 210
160 IF T>A(S) THEN 190
170 U=S
180 GOTO 130
190 L=S
200 GOTO 130
210 PRINT "ITEM NOT IN
    LIST"
220 GOTO 250
230 S=U
240 PRINT "ITEM FOUND IN
    POSITION";S
250 .....
```

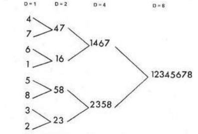


Fig.27. The 'Merge-Sort' process.

Earlier we said that we would look at the need for flowchart box 2 in Fig. 26, and we will do this now with reference to the above program. We will dry-run it with U=3 and L=1 giving us a list of 3 items. (Say A(1)=10, A(2)=12 and A(3)=52). Program line 120 above corresponds to box 2 in Fig.26 and we will omit it mentally and see what happens when we input a value fo 52 for T in line 110 above. Line 130 assigns a value of 2 to S (L+U=4, 4/2=2, INT(2)=2, S=2). T is not equal to 12, the value of A(2) in line 40 and so we move on to line 150. U is not equal to L, and so on to 160. T is greater than A(2) which shows that if T is contained in the list at all it must be in the upper half.

We now set L to S (line 190) which gives L=2, U=3. Line 200 takes us back to line 130 where a new value of S is

calculated. (U+L=5, 5/2=2.5, INT(2.5)=2, S32).
Now, T is not equal toA(S) — line 140
U is not equal to L — line 150
and T is greater than A(S) — line 160
so we make L=S (which it already is) and branch back to 130. Now we see the problem. The INT function used in line 130 to calculate the mid-position of U and L will only round down to the nearest integer — it cannot round up — and consequently we can never look at the last item in the list to see if it containd T. Obviously, then, when the last list position does contain T the algorithm would not terminate without the inclusion of some test (Fig.26, box 2 for example) to see if T were contained in the last list position.

EFFICIENT ALGORITHM

You remember that the binary search algorithm is much more efficient than a simple search, but it suffers from the drawback that the list to be searched has to be in ascending numerical order. The process of sorting a list into order can in itself be very lengthy, especially if we use the simple sort routine given earlier in the series. Fortunately there is a sort routine which is very efficient and is based on the merge of two sorted lists that we saw in a previous article. If you can imagine an unsorted list of eight items then this algorithm would take each of the 4 consecutive pairs of numbers in this list in turn and perform a two list merge on them which will give four pairs of numbers each of which will be in numeric order. The algorithm then takes the first two pairs thus generated and merges them to form a sorted list of four numbers and then takes the second two pairs and merges them also.

We now have two sets of four numbers, each set being in numerical order. The final process is to merge these two lists of four items into one list of 8 items and the sort is complete (see Fig.27). The flowchart of this algorithm is given as Fig.28.

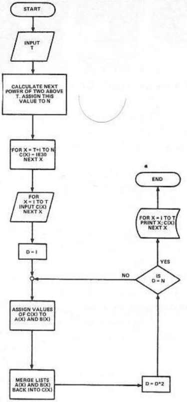

START
INPUT T
CALCULATE NEXT POWER OF TWO ABOVE T. ASSIGN THIS VALUE TO N
FOR X = T+1 TO N C(X) = 1E30 NEXT X
FOR X = 1 TO T INPUT C(X) NEXT X
D = 1
ASSIGN VALUES OF C(X) TO A(X) AND B(X)
MERGE LISTS A(X) AND B(X) BACK INTO C(X)
D = D*2
IS D = N
NO
YES
FOR X = 1 TO T PRINT X; C(X) NEXT X
END

Fig.28. The 'Final Programme' flowchart.

```
  5 REM= =MERGE SORT= =
 10 PRINT "HOW MANY
ITEMS TO BE SORTED"
 20 INPUT T
 30 A=1
 40 IF A>T THEN 70
 50 A=A*2
 60 GOTO 40
 70 N=A
 75 DIMAA(A/2+1),
    B(A/2+1),C(A+1)
 80 FOR X=1+1 TO N
 90 C(X)=1E30
100 NEXT X
110 PRINT "INPUT VALUES
    TO GO IN LIST"
120 FOR X = 1 TO T
130 PRINT X
140 INPUT C(X)
150 NEXT X
160 PRINT "SORT
    BEGINS NOW"
170 D=1
180 A=1
190 B=1
200 C=1
210 FOR X = 1 TO T
220 B(B)=C(C)
230 B=B+1
240 C=C+1
250 NEXT X
260 FOR X=1 TO D
270 A(A)=C(C)
280 A=A+1
290 C=C+1
300 NEXT X
310 IF C<>N+1 THEN 210
320 A=1
330 B=1
340 C=1
350 S=A-1
360 IF A(A)>B(B) THEN 400
370 C(C)=A(A)
380 A=A+1
390 GOTO 420
400 C(C)=B(B)
410 B=B21
420 C=C+1
430 IF A=S+D+1 THEN 520
440 IF B=S+D+1 THEN 460
450 GOTO 360
460 FOR X = A TO S+D
470 C(C)=A(A)
480 C=C+1
490 A=A+1
500 NEXT X
510 GOTO 570
520 FOR X=B to S+D
530 C(C)=B(B)
540 C=C+1
550 B=B+1
560 NEXT X
570 IF A+B<N+2 THEN 350
580 D=D *2
590 IF D<>N THEN 180
600 PRINT "THE SORTED
    LIST IS"
610 FOR X=1 TO T
620 PRINT "      ";X;C(X)
630 NEXT X
640 END
```

The program starts off by taking an input to the variable T. This is used to tell the program how many items are to be sorted. The next part of the program (lines 30 to 70) calculates the next power of 2 above the input value (T). This value is assigned to the variable N and is the actual number of items that will be sorted. Next items, T+1 To N, are made very large (1E30) so that after sorting they will still occupy positions T+1 TO N in list C(X). The sort now begins. The "INPUT" list C(X) is first split up into two lists A(X) and B(X) by lines 210-310. The variable D is used to indicate the number of values in A(X) and B(X) that are to be merged in each step (see Fig.27).

The first D (initially D=1) items in C(X) are assigned to the first D items in A(X), then the second D items in C(X) are assigend to the first D items in B(X), then the third D items in C(X) are assigned to the second D items in A(X).......then B(X).....then A(X) and so on.

Simon Goodwin

SNIPER

Outwit the malevolent robots in this all-action game for the Genie or TRS-80 Model 1.

Here is a game to pit your wits and reflexes against security robots at the top secret research headquarters of IBM (Intergalactic Business Mogul). You are looking through the giant mainframe computer-room, in search of interesting secrets, when you hear a rumbling noise from between the rows of processors and disc units . . .

You dive out of the way just in time as a hi-blast quarkon grenade rolls past you and detonates nearby. In the dimness you can just make out the glowing sensors of a security robot at the other end of the room. The robot's cybo-grip unit reaches behind and it primes another grenade, ready to roll it towards you. The bomb explodes prematurely, showering blue paint over the robot. Unperturbed, the security robot takes aim again.

You must disable the robot and any others which come to its assistance. To do this you are armed with a lightweight, high-resolution, laser rifle, but it is difficult to hit the robot as it dodges back and forth in the darkness at the other end of the computer room. Alternatively you could brave the grenades and attempt to sneak up and eliminate the robot with a sharp blow to its sensitive omniwave radiation sensors. The decision is yours; but hurry, because here comes another grenade . . .

Use the arrow keys or control keys to manoeuvre the sniper around the computer room. Press the space key to fire your laser rifle. You will lose a point every time you shoot (since you are wasting ammunition and drawing attention to yourself) and you lose a 'life' if you collide with a grenade. Points are awarded each time you sucessfully shoot the robot (the shorter the range the greater the score) and if you manage to 'mug' the robot and destroy its sensors. Be warned — you will be unable to shoot down the grenades with your rifle, and the robot will try to evade you if you creep up on it.

HINTS ON CONVERSION

SNIPER is written in Microsoft 12K Z80 BASIC. As it is a graphic game a number of conversions must be made to enable it to be run on machines not compatible with the TRS-80 Level 2. The game uses the PRINT @ facility to place text and graphics on the screen. The TRS-80 display consists of 1024 characters. The character-code for each element is stored in memory between address 15360 and 16383. Consequently PEEK (15360) will return the ASCII code of the character in the left most position of the top line. PEEK (15487) does the same for the last character of the second

LINE FUNCTION

Lines 10-30	Identify the program and send the computer to the set-up routine at line 450.
Line 40	Introduces the routine for rolling bombs across the display. These lines form the main part of the program and consequently they have been placed at its head, where BASIC can find them quickly. The game would be about a third slower if the 'roll bombs' routine was at the end of the program.
Line 50	Selects the next bomb to be moved.
Lines 60-130	Check that the bomb has not yet hit the end wall of the computer room. If it has done so then a new bomb is launched.
Lines 140-160	Advance the bomb across the display.
Line 170	Checks to see whether or not the bomb has spontaneously exploded.
Lines 180-290	Reduce the number of lives if the bomb has hit the player.
Lines 300-430	Handle the hi-blast quarkon grenade explosions.
Lines 440-490	Part of the set-up routine for the game. They clear variable storage and the TRS-80 display. Lines 450 and 460 will not be needed on most computers. Adjust the value of BMAX to change the maximum number of bombs in action at any time. Its value must be a power of two (e.g. 1, 2, 4, 8 or 16).
Lines 500-560	Sets up the graphic characters that will be used in the game. Each item is assumed to be one character high and three characters wide.
Lines 570-680	Draw the walls of the computer room and populate it with chest freezer shaped machines.
Lines 690-730	Prepare the main recording variables of the game. These are P (player's position on screen for PRINTing), E (enemy's position), SC (score), B (bomb position) and LV (number of

line, and so on.

Each character-position has a number which is used as a reference for the PRINT @ command. The top left character position is numbered 0, through to 63 at the end of the line 1023 at the end of the screen. The display consists of 16 lines of 64 characters. For example PRINT @960, "SNIPER"; would cause the word to be displayed at the left-hand side of the last line of the screen.

The STRING$(X,Y) function returns X copies of the character with ASCII code Y. An equivalent function is available in BBC BASIC — on other machines you will have to set up a loop to generate the appropriate string using the CHR$ function.

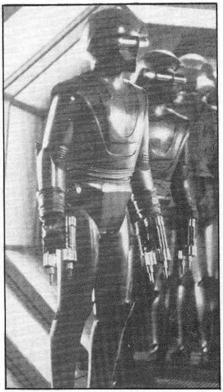
SNIPER makes considerable use of the logical AND operation. This is available on most micros, but notably not the Apple or current Sinclair machines. A logical AND is used to tell the computer to ignore some of binary digits in a value. The value 30 is written 11110 in binary, and the value 19 is 10011, so that the expression 30 AND 19 has the binary value 100,10 or 18 decimal. (The 1 is copied in the columns where it appears in both numbers). Type PRINT 30 AND 19 on your computer to see if this feature is supported.

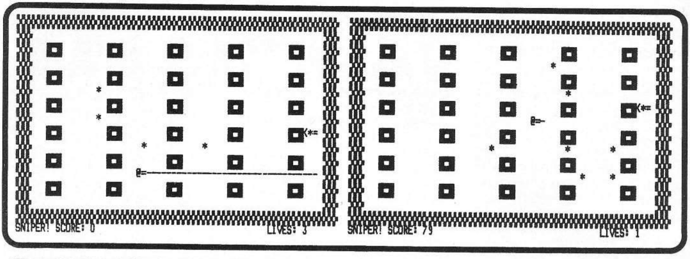
The program uses the AND function to generate random numbers. For example RND(3) generates (at random) 1,2 or 3. Long names are used for some string variables (e.g. PLAYER$) but these can be shortened to their first two characters (PL$) without ill-effect.

PEEK(14400) is used to poll the computer's keyboard. The game is controlled by pressing five keys. On a Video Genie you may use ESC and CTRL to move the sniper up and down the screen. If the keys are held down the sniper will move steadily. The CLEAR and TAB keys can be used to move left and right, the SPACE key cause a shot to be fired. If your computer has arrow-keys in place of ESC and CTRL, or CLEAR and TAB are not adjacent on your keyboard, then you can modify the program to recognise control from the arrows simply by changing line 900 so that it starts 'IF (KB AND 32) . . .'.

Sound-effects are generated as SNIPER runs. Program lines containing the OUT instruction are used to send noises to the Genie or TRS-80 cassette interface. These lines won't work on other computers but they may be missed out without harming the game.

	remaining lives).
Lines 740-760	Set the starting position of all bombs.
Lines 770-790	Label the display.
Lines 800-870	Form the main loop of the game. For simplicity it has been written in four parts. One part rolls the grenades, another allows the player to move, the third handles rifle-fire, and the final part controls the robot's movement.
Lines 880-940	Examine the keyboard (see 'Hints on Coversion') and adjust the player's position accordingly.
Lines 950-970	Punish the sniper if a bomb has been trodden upon.
Lines 980-1000	Blank out the old display of the sniper and set up the new one.
Line 1010	Checks whether or not the sniper has jumped on top of the robot. If so lines 1030-1110 modify the score and display accordingly.
Lines 1120-1190	Decide the robot's move. Line 1160 selects a new direction of movement (up or down) to home in upon the player.
Lines 1200-1210	Adjust the robot's position, preventing moves off the screen!
Lines 1220-1260	Set up the new display of the robot, blanking out the old one as required.
Lines 1270-1290	Handles rifle fire. Line 1280 determines whether or not the space key (fire) has been pressed and line 1290 excludes alternate lines of the display from use (those military-spec. cabinets are 100 per cent laser-proof!).
Lines 1300-1370	Display the laser-beam with an appropriate sound-effect. Line 1300 works out its length and 1370 checks whether or not the robot has been hit.
Line 1380	Allocates points in accordance with the distance between the sniper and the robot.
Lines 1390-1480	Illustrate the robot explosion in sound and vision.





SNIPER! SCORE: 0 LIVES: 3

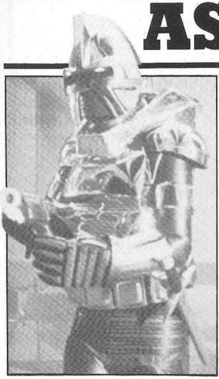
SNIPER! SCORE: 79 LIVES: 1

```
10 REM ** TRS-80 SNIPER
20 REM ** Simon Goodwin
30 GOTO 450
40 REM ** Roll the bombs!
50 C = C + 1 AND BMAX - 1
60 IF (B(C) AND 63) > 3 THEN 140
70 IF RND(2) < 1 OR (E AND 64) < 64 THEN RETURN
80 B(C) = 54 + (E AND 960)
90 IF Q = B(C) THEN 420
100 Q = B(C)
110 PRINT @B(C),BOMB$;
120 IF Q=P THEN 200
130 RETURN
140 PRINT @B(C),BLANK$;
150 B(C) = B(C) - 3
160 IF (B(C) AND 63) > 3 THEN PRINT @B(C),BOMB$;
170 IF RND(12) > 1 THEN IF B(C) <> P THEN RETURN
180 REM ** See where bomb has gone
190 IF B(C) <> P THEN 310
200 LV = LV - 1
210 PRINT @1016,LV;
220 LP = 515
230 IF LV > 0 THEN 260
240 PRINT @984,"G A M E   O V E R !";
250 GOTO 250
260 IF KB<900 THEN 290
270 KB = KB + B(C)
280 B(C) = LP
290 LP = 515
300 REM ** Bomb explosion
310 FOR Y = 0 TO 1
320 OUT 255,1
330 PRINT @B(C),"+!+";
340 OUT 255,2
350 OUT 255,1
360 PRINT @B(C),"   ";
370 OUT 255,2
380 NEXT Y
390 IF KB < 900 THEN 420
400 B(C) = KB - 1100
410 RETURN
420 B(C) = 515
430 RETURN
440 REM ** SNIPER Program start
450 CLEAR 500
460 DEFINT A-Z
470 BMAX=8
480 DIM B(BMAX-1)
490 CLS
500 REM ** Set up graphics
510 LUMP$ = CHR$(191) + CHR$(179) + CHR$(191)
520 WALL$ = STRING$(3,153)
530 BLANK$ = STRING$(3,32)
540 PLAYER$ = "@=-"
550 ENEMY$ = "<*="
560 BOMB$ = CHR$(32) + CHR$(42) + CHR$(32)
570 REM ** Draw the room
580 PRINT @0,STRING$(64,153);
590 PRINT @896,STRING$(64,153);
600 FOR Y = 64 TO 840 STEP 64
610 PRINT @Y,WALL$;
620 PRINT @Y + 61,WALL$;
630 NEXT Y
640 FOR Y = 134 TO 800 STEP 128
650 FOR X = 0 TO 50 STEP 12
660 PRINT @X + Y,LUMP$;
670 NEXT X
680 NEXT Y
690 REM ** Set up the variables
700 P  = 515
710 E  = 505
720 SC = 0
730 LV = 3
740 FOR Y=0 TO BMAX-1
750 B(Y) = 194
760 NEXT Y
770 PRINT @1010,"Lives:";LV;
780 PRINT @P,PLAYER$;
790 PRINT @960,"SNIPER! Score:";SC;
800 REM ** Main loop of game
810 GOSUB 890 ' Player's move
820 GOSUB 40 ' Roll bomb 1
830 GOSUB 1280 ' Fire rifle
840 GOSUB 40 ' Roll bomb 2
850 GOSUB 1130 ' Target's move
860 GOSUB 40 ' Roll bomb 3
870 GOTO 810
880 REM ** Check keys and move player
890 KB = PEEK(14400)
900 IF (KB AND  2) THEN IF (P AND 63) > 5  THEN LP = P - 3
910 IF (KB AND 64) THEN IF (P AND 63) < 56 THEN LP = P + 3
920 IF (KB AND  8) THEN IF P > 128 THEN LP = P - 64
930 IF (KB AND 16) THEN IF P < 896 THEN LP = P + 64
940 IF P = LP OR PEEK(LP + 15360) > 128 THEN RETURN
950 IF PEEK(LP + 15361) <> 42 THEN 980
960 KB = 1100
970 GOTO 200
980 PRINT @P,BLANK$;
990 PRINT @LP,PLAYER$;
1000 P = LP
1010 IF P <> E THEN RETURN
1020 REM ** You have 'mugged' the enemy
1030 SC = SC + 10
1040 FOR Y = 0 TO 7
1050 OUT 255,1
1060 PRINT @E,")-(";
1070 OUT 255,2
1080 PRINT @E,ENEMY$;
1090 NEXT Y
1100 LP = 515
1110 GOTO 1390
1120 REM ** Move the enemy
1130 L = L - 1
1140 IF L < 6 AND L > 0 THEN RETURN
1150 IF L > 5 THEN 1200
1160 IF E < P THEN D=64 ELSE D=-64
1170 IF E + D < 64 OR E + D > 896 THEN D=-D
1180 IF L < 1 THEN L = RND(4) + 7
1190 RETURN
1200 NE = E + D
1210 IF NE < 128 OR NE > 832 THEN D = -D
1220 PRINT @E,BLANK$;
1230 PRINT @NE,ENEMY$;
1240 E = NE
1250 IF E = P THEN 1030
1260 RETURN
1270 REM ** Check for rifle fire
1280 IF (KB AND 128) = 0 THEN RETURN
1290 IF (P AND 64) < 64 THEN RETURN
1300 L = 57 - (P AND 63)
1310 PRINT @P + 3,STRING$(L,45);
1320 FOR Y = 0 TO 12
1330 OUT 255, Y AND 3
1340 NEXT Y
1350 PRINT @P + 3,STRING$(L,32);
1360 IF SC > 0 THEN SC = SC - 1
1370 IF (P AND 960) <> (E AND 960) THEN 1470
1380 SC = SC + (P AND 63)/3
1390 FOR Y = 0 TO 12
1400 PRINT @E,"=+=";
1410 OUT 255,Y AND 3
1420 PRINT @E,"0+0";
1430 NEXT Y
1440 PRINT @E,BLANK$;
1450 E = 505
1460 PRINT @E,ENEMY$;
1470 PRINT @974,SC;
1480 RETURN
```

Chris Palmer

SARDAUKAR ASSAULT

Defend the Golden Tower against hoards of attacking space warriors in this Atari game.



Trapped on a remote outpost somewhere at the edge of the galaxy you are the last remaining defender of your empire. Materialising above your base are scout ships of the dreaded Sardaukar fleet. Can you destroy them before they blast through the golden tower and damage your power base? If you succeed in wiping out one wave, then an even stronger fleet is sent against you. See how many of them you can destroy before you are overrun.

Sardaukar Assault is written in BASIC for the Atari 400/800 with a minimum memory of 16K.The game is controlled by a joystick plugged into the lefthand port, but can be easily converted to use the keyboard.

When the game starts, the joystick controls the movement of a gold coloured dot, which is the point to which your defence laser will fire. To destroy the alien attackers you have to hit them with the laser. The game ends when the Sardaukar's laser shots have penetrated the green power base at the bottom of the tower. After each successive wave your tower is rebuilt. Sardaukar Assault is written using Atari BASIC, which is quite like most forms of Microsoft, so the BASIC in the program should be easily convertible. It is run in graphics mode 7+16, which gives the computer a screen resolution of 160 x 96 in 4 colours. The graphics commands used are: PLOT, which turns on a single dot, DRAWTO, which draws a line from the current PLOT or DRAWTO position to the specified location, SETCOLOR, which primes a colour register with a colour and a luminosity, COLOR, which changes the PLOT or DRAWTO colour and LOCATE, which reads the value of a dot into a specified variable.

All the commands should be able to be duplicated on another machine providing it has a PLOT or DRAWTO equivalent. The one feature of the Atari that you might have trouble in duplicating is that if you change the colour or luminance in a SETCOLOR statement, then everything that has been plotted in that colour now assumes the new colour. On the BBC computer this can be achieved by using the VDU19 command.

To convert the game to run with the keyboard, change line 2010 to a keyboard GET statement and then change lines 2020-2055 to alter the values of X and Y according to which key has been pressed.

LINE FUNCTION

10-25	Sets up the position array and clears all its positions.
30-90	Prints up beginning screen and rotates the background colours.
100-130	Sets up specific variables, changes the graphics mode and sets the colours.
135-137	Goes to the routines to print the mountains and the golden tower. Also reads the joystick.
140-155	Tests for the fire button. Checks to see whether a ship has been encountered and checks to see whether all the ships have been cleared.
157-175	Plots laser track from base.
180	Plots laser aiming position.
185-200	Random jumps for alien fire and alien appearance.
999	End of main program loop.
1000-1050	Draws mountains.
1100-1180	Draws golden tower and power base.
2000-2110	Reads joystick and examines new position.
3000-3080	Plots Sardaukar ship randomly on screen.
4000-4100	Checks to see if position is occupied by an active alien and if so explodes it.
5000-5090	Changes graphic mode, prints wave number and score so far.
6000-6110	Plots Sardaukar shots from random ship position and checks to see if power base has been reached.
7000-7100	Explodes base and prints end of screen display.
7200-7260	Provides screen flash and wipes out base.

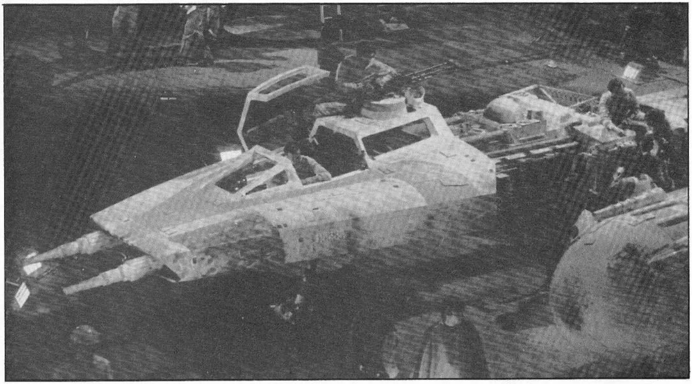
►

```
10 REM SARDAUKAR ATTACK
20 CLR :DIM P(20,2):KILL=4
25 FOR I=0 TO 20:FOR T=1 TO 2:P(I,T)=0:N
EXT T:NEXT I
30 GRAPHICS 2+16
40 POSITION 16,0
50 PRINT #6,"SARDAUKAR"
60 POSITION 17,2
70 PRINT #6,"ASSAULT"
75 FOR P=1 TO 3
80 FOR I=1 TO 14:SETCOLOR 4,I,10:SOUND P
,I*10+P,10,12:FOR T=1 TO 50:NEXT T:NEXT
I
90 NEXT P
100 REM SET UP VARIABLES
110 Z=1:X=80:CO=0:PL=1
120 GRAPHICS 7+16
130 SETCOLOR 4,0,0:SETCOLOR 0,1,12:SETCO
LOR 1,6,12:SETCOLOR 2,11,12
135 GOSUB 1000
137 GOSUB 2000
140 IF STRIG(0)<>0 THEN 175
150 IF L<>0 THEN GOSUB 4000
155 IF CO>=KILL THEN GOTO 5000
157 SOUND 0,150,8,15
160 COLOR 1:PLOT 80,60:DRAWTO X,Y
165 SOUND 0,200,8,15
170 COLOR 0:PLOT 80,60:DRAWTO X,Y
175 SOUND 0,0,0,0
180 COLOR 1:PLOT X,Y:COLOR 0:PLOT X,Y
185 O=INT(RND(1)*20+1)
190 IF O>16 AND O<18 THEN GOSUB 3000
200 IF O<15 AND O>8 THEN GOSUB 6000
999 GOTO 137
1000 REM DRAW MOUNTAINS
1005 PLOT 1,75
1010 FOR I=1 TO 159 STEP 5
1020 Y=INT(RND(1)*20+76)
1030 COLOR 2
1040 DRAWTO I,Y
1050 NEXT I
1100 REM DRAW BASE
1105 COLOR 1
1110 FOR I=70 TO 80
1120 PLOT I,95:DRAWTO I,96-(I-68)^2/4
1130 PLOT 160-I,95:DRAWTO 160-I,96-(I-68
)^2/4
1140 NEXT I
1145 COLOR 3
1150 FOR Y=90 TO 95
1160 PLOT 78,Y:DRAWTO 82,Y
1170 NEXT Y:Y=50
1175 FOR P=1 TO 3:SOUND P,0,0,0:NEXT P
1180 RETURN
2000 REM READ JOYSTICK
2010 S=STICK(0)
2020 IF S=7 THEN X=X+2
2030 IF S=11 THEN X=X-2
2040 IF S=13 THEN Y=Y+2
2050 IF S=14 THEN Y=Y-2
2055 IF S=15 THEN RETURN
2060 IF X>159 THEN X=159
2070 IF X<1 THEN X=1
2080 IF Y>70 THEN Y=70
2090 IF Y<1 THEN Y=1
2100 LOCATE X,Y,L
2110 RETURN
3000 REM PLOT SARDAUKAR
3010 IF PL>KILL THEN RETURN
3020 A=INT(RND(1)*140+5)
3030 B=INT(RND(1)*20+5)
3040 P(Z,1)=A:P(Z,2)=B:Z=Z+1:IF Z>20 THE
N 7000
3050 COLOR 3:FOR P=1 TO 6
3060 PLOT P/2+A,B+P:DRAWTO (A+6)-P/2,B+P
3065 SOUND 0,200,10,7+P:SOUND 1,201,12,7
-P
3070 NEXT P:PL=PL+1
3075 SOUND 0,0,0,0:SOUND 1,0,0,0
3080 RETURN
4000 REM ALIEN HIT
4010 FOR I=1 TO Z
4020 IF X>=P(I,1) AND X<P(I,1)+6 AND Y>=
P(I,2) AND Y<P(I,2)+6 THEN 4040
4030 NEXT I:RETURN
4040 A=P(I,1):B=P(I,2)
4060 COLOR 0:FOR P=1 TO 6
4070 PLOT P/2+A,B+P:DRAWTO (A+6)-P/2,B+P
4075 SOUND 0,200-P*20,10,15
4080 NEXT P
4090 P(I,1)=0:P(I,2)=0:SC=SC+10:CO=CO+1
4095 SOUND 0,0,0,0
4100 RETURN
5000 REM END OF WAVE
5010 GRAPHICS 1+16:WA=WA+1
5020 POSITION 0,0
5030 PRINT #6;"WAVE ";WA;" COMPLETED"
5040 PRINT #6;"SCORE = ";SC
5045 KILL=KILL+2:CO=0:PL=1:FT=0
5050 FOR T=1 TO 1000:NEXT T
5060 FOR I=1 TO 20:FOR T=1 TO 2
5070 P(I,T)=0
5080 NEXT T:NEXT I
5090 GOTO 100
6000 REM SARDAUKAR FIRE
6010 FP=INT(RND(1)*Z+1)
6015 IF P(FP,1)=0 THEN RETURN
6020 COLOR 2
6025 SOUND 0,100,6,15
6030 PLOT P(FP,1)+3,P(FP,2)+5
6040 DRAWTO 80,62+FT
6045 SOUND 0,50,6,15
6050 COLOR 0
6060 PLOT P(FP,1)+3,P(FP,2)+5
6070 DRAWTO 80,62+FT
6100 FT=FT+1:IF FT>28 THEN 7000
6105 SOUND 0,0,0,0
6110 RETURN
7000 REM END
7010 FOR I=5 TO -10 STEP -0.5
7020 FOR T=1 TO 14
7025 SOUND 0,T+200-I*2,10,T
7030 SETCOLOR 0,1,T
7040 FOR P=1 TO I:NEXT P:NEXT T:NEXT I
7045 SOUND 0,0,0,0:GOSUB 7200
7050 GRAPHICS 1+16
7060 POSITION 0,0:PRINT #6;"GAME OVER"
7070 PRINT #6;"SCORE = ";SC
7080 PRINT #6;"PRESS FIRE TO START"
7090 IF STRIG(0)<>0 THEN 7090
7100 RUN
7200 FOR I=0 TO 14:SETCOLOR 4,0,I:NEXT I
:SOUND 0,10,4,15
7210 FOR I=14 TO 0 STEP -1
7215 SOUND 0,10,4,I
7220 SETCOLOR 4,0,I:FOR T=1 TO 3*I
7230 NEXT T:COLOR 0
7240 PLOT 70,90-I:DRAWTO 90,90-I
7250 NEXT I
7260 RETURN
```

Eric Smith

SPECTRUM ZAP

Anihilate aliens as they descend from the skies in this fast-moving Spectrum game.



At the start of this game you are given three lives, with your laser base positioned at the bottom of the screen.

The alien appears at the top of the screen and moves down towards your base line in one of three directions; either straight down, vertically left, or vertically right.

The screen is wrapped round from the alien's point of view, but not from the player's. You fire at the alien by pressing the '0' (zero) key and move left and right by pressing the '5' and '8' keys respectively. If, however, on your third shot you miss the alien then it can change direction, but will still approach your base line.

If the alien touches your base line then you lose one of your lives (if you are on your last life the game ends). When you destroy the alien then you gain up to 2000 points, depending on how far down the screen it was when your laser gun obliterated it.

Throughout the game 'Mystery Ships' will wander across the screen. If you hit them then you gain up to 10,000 points and a free life. (But you have to hit them in exactly the right place for the laser bolt to take effect!!!). You are allowed only three shots at them then your laser base becomes ineffective and the ship passes across the screen whereupon you lose a life, (this only applies if you have one life to spare: ie missing a ship cannot put you out of the game.)

The game ends when three aliens land. However, if you get over 100,000 points the aliens wrath rises and they start to descend not from the top of the screen, but slightly further down so as to catch you out at the wrong side of the screen.

Throughout the game sound effects are produced as the alien descends, when you fire your laser bolt, when you hit the alien, when the alien lands, when the 'Mystery Ship' appears and when the game ends.

The program runs quite quickly even though the program contains no machine code. The 'Mystery Ship' routine was written out with the main printing loop as a separate routine so that the speed of the program was not impaired. The highest score obtained so far is 110,349 before three aliens landed. Good shooting!

HINTS ON CONVERSION

This program should be able to be converted onto most machines that allow text and high resolution graphics to be mixed onthe same screen. This is necessary because the laser shots are created using high resolution line graphics while the laser base and invaders are made up of user defined graphics characters. These are placed on the screen by using a PRINT AT statement and can be replaced by a screen POKE on any machine that doesn't support this, but does have a memory mapped screen.

▶

The screen on the Spectrum is 255 pixels by 175 pixels with the origin (0,0) in the bottom left corner. The character screen is 32 × 22 with the origin for PRINT AT commands at the top left of the screen. It is worth noting that the Spectrum's PLOT and DRAW statements work differently to most machines. If you want to PLOT at a point (50,50) and then DRAW to the point (100,100) you would have to use the form PLOT 50,50 : DRAW 50,50. This is because the co-ordinates in the DRAW statement are offsets from the PLOT statement, not a direct reference to a screen location.

If you have a machine that is capable of producing definable graphics the conversion of the Spectrum's definable characters is simple. The BIN statements in lines 20-70 contain all the information for bit patterns of the characters, expressed in binary form. For your own machine either convert them into their decimal equivalents or leave them as binary, depending on which method your machine uses.

LINE FUNCTION

Lines 1-3	Produce the instructions if necessary.
Lines 5-70	Produces the user definable space invader, spaceship explosion and laser base.
Lines 100-115	Reset the game.
Lines 120-160	Reset the screen for another alien
Lines 170-280	Are the main screen print and the exit to the fire routine.
Lines 400-490	Are the fire routine which draws and undraws the laser bolt. This also provides the exit to the scoring routine.
Lines 500-540	Contains the scoring routine and the exit to the 'Mystery Ship' routine. It also increases the print position of the alien if the score is over 100,000 and RND is greater than 0.7.
Lines 600-895	Produces the whole 'Mystery Ship' routine: ie the moving of the spaceship and the base, the firing of the laser base and the explosion the points scored and the free life.
Lines 900-940	Deals with the loss of a life if the spaceship was missed.
Lines 980-990	Ends the game.
Lines 1000-1050	Produce the random movement for the alien.
Lines 1100-1110	Produce the effect of the wrapped round screen.
Lines 2000-2160	Are the instructions.

```
   3 GO SUB 2000
   5 INK 9: BORDER (RND*7): PAPE
R (RND*7): CLS
  10 FOR k=1 TO 6: FOR n=0 TO 7:
 READ a: POKE USR (CHR$ (143+k))
+n,a: NEXT n: NEXT k
  20 DATA BIN 00011111,BIN 00100
000,BIN 01111111,BIN 10100100,BI
N 10100100,BIN 01111111,BIN 0010
0000,BIN 00011111
  30 DATA BIN 11111111,BIN 0,BIN
 11111111,BIN 10010010,BIN 10010
010,BIN 11111111,BIN 0,BIN 11111
111
  40 DATA BIN 11111000,BIN 00000
100,BIN 11111110,BIN 00100101,BI
N 00100101,BIN 11111110,BIN 0000
0100,BIN 11111000
  50 DATA BIN 00111100,BIN 01111
110,BIN 01011010,BIN 01111110,BI
N 00111100,BIN 01000010,BIN 1000
0001,BIN 10000001
  60 DATA BIN 10000001,BIN 01011
010,BIN 00100100,BIN 01000010,BI
N 01000010,BIN 00100100,BIN 0101
1010,BIN 10000001
  70 DATA BIN 00010000,BIN 00010
000,BIN 00010000,BIN 00010000,BI
N 0101000,BIN 01111100,BIN 01111
100,BIN 11111111
 100 LET life=3
 105 LET s=2
 110 LET sc=0
 115 LET a=19: LET b=13
 120 PRINT AT 0,25;"     ";AT 0,
25;"↑↑↑↑↑"( TO life-1); FLASH 1;
AT 0,0;"SCORE ";sc
 130 FOR z=0 TO 31: PRINT AT 19,
z;"_": NEXT z
 150 LET c=s: LET d=INT (RND*22)
+4
 160 GO SUB 1000
 170 PRINT AT a,b;"_F_"
 180 PRINT AT c,d;" D ";AT c-1,d
;"   "
 190 BEEP 0.005,c
 200 LET b=b+(INKEY$="8" AND b<2
8)-(INKEY$="5" AND b>0)
 210 IF INKEY$="0" THEN GO TO 400
 220 IF fr=3 THEN GO SUB 1000
 230 LET d=d+m
 240 LET c=c+1
 250 IF d<=2 THEN GO SUB 1100
 260 IF d>=28 THEN GO SUB 1110
 270 IF c>=20 THEN GO TO 900
 280 GO TO 170
 400 LET w=8*(b+1)+3
 410 LET fr=fr+1
 420 PLOT w,24
 430 LET q=(18-c)*8
 440 DRAW 0,q
 450 PLOT OVER 1;w,q+24
 460 DRAW OVER 1;0,-q
 470 FOR l=20 TO 25: BEEP .005,l
: NEXT l
 480 IF b=d THEN GO TO 500
 490 GO TO 220
 500 PRINT AT c,d;" E ": FOR l=4
0 TO 50: BEEP .01,l: NEXT l: PRI
NT AT c,d;"   "
 510 LET sc=sc+INT (99.9*c): PRI
NT AT 0,0; FLASH 1; BRIGHT 1;"SC
ORE ";sc
 520 IF RND>.95 OR RND>.85 AND s
c<80000 OR RND>.7 AND sc<20000 O
R RND>.5 AND sc<3000 THEN GO TO
600
 530 LET s=s+(sc>100000 AND RND>
.7)
 540 GO TO 120
 600 FOR z=0 TO 31: PRINT AT 19,
z;"_": NEXT z
 610 LET cs=0
 640 LET u=2: LET t=0
 650 PRINT AT u,t;" ABC";AT a,b;
"_F_"
 670 BEEP 0.05,t+20
 680 LET b=b+(INKEY$="8" AND b<2
8)-(INKEY$="5" AND b>2)
 690 IF INKEY$="0" AND cs<3 THEN
 GO TO 730
 700 LET t=t+1
 710 IF t>=27 THEN GO TO 890
 720 GO TO 650
 730 LET e=8*(b+1)+3
 740 LET cs=cs+1
 750 LET ans=INT (RND*3)+1
 760 PLOT e,24
 770 LET q=16*8
 780 DRAW 0,q
 790 PLOT OVER 1;e,q+24
 800 DRAW OVER 1;0,-q
 810 FOR l=20 TO 25: BEEP 0.005,
l: NEXT l
 820 IF b=t+ans THEN GO TO 840
 825 LET t=t+1
 830 GO TO 650
 840 PRINT AT u,t;"EEEEE"
 850 LET sc=sc+INT (RND*10000)
 860 LET life=life+1*(life<5)
 870 PRINT AT u,t;"     "
 880 GO TO 120
 890 LET xb=8*(b+1)+3: LET yb=15
2: PLOT 240,155: DRAW (xb-240),(
-142): FOR l=1 TO 50: NEXT l: DR
AW OVER 1;(240-xb),(142): PRINT
AT u,t;"     ": LET life=life-(l
ife>1)
 895 GO TO 920
 900 PRINT FLASH 1;AT 10,10;"ALI
EN LANDED": FLASH 0
 910 LET life=life-1
 920 IF life=0 THEN GO TO 980
 930 FOR l=1 TO 15: BEEP 0.1,(RAN
D*20)+5: NEXT l
 940 CLS : GO TO 120
 980 PRINT AT 10,10;"GAME OVER,
";AT 12,12; FLASH 1;"A L I E N
S": FLASH 0
 990 FOR l=10 TO 50: BEEP .1,l:
NEXT l: STOP
1000 LET rnd=RND
1010 IF rnd<0.333 THEN LET m=-1
1020 IF rnd>=0.333 AND rnd<=0.66
6 THEN LET m=0
1030 IF rnd>=0.666 THEN LET m=1
1040 LET fr=0
1050 RETURN
1100 PRINT AT c-1,d;"  ": LET d
=27: RETURN
1110 PRINT AT c-1,d;"  ": LET d
=3: RETURN
2000 PRINT BRIGHT 1; FLASH 1;AT
5,10;"A L I E N S": BRIGHT 0: PR
INT FLASH 1;AT 10,0;"Do you want
 the instructions ?";AT 11,12;"(
Press y/n)": FLASH 0
2010 IF INKEY$="n" THEN RETURN
2020 IF INKEY$="y" THEN GO TO 20
40
2030 GO TO 2010
2040 CLS
2050 PRINT "In the game of ALIEN
S the objectis to blast as many
aliens off  the screen before th
ey touch theground.
                You move right by pr
essing the  '8' key and left by
pressing the'5' key, to fire you
r laser you must press the "0" k
ey."
2060 PRINT "The aliens come in t
hree          directions: straight
 down,        vertically right and
 vertically left. Up to 2000 poi
nts may be  scored by hitting th
e aliens,   however if on your t
hird laser  bolt you miss the al
ien then thealien may change dir
ection."
2070 PRINT : PRINT FLASH 1;"Pres
s ENTER to continue.": FLASH 0
2080 IF CODE INKEY$<>13 THEN GO
TO 2080
2090 CLS : PRINT "Each time an a
lien touches the  base line then
 one of your livesdisappears. Th
e number of lives you have in re
serve is the          number of arro
ws at the top         right hand sid
e, your score is  on the top lef
t hand side.          On the third a
lien landing the  game ends suit
ably."
2100 PRINT "A 'Mystery Ship' wil
l appear to move across the scre
en. If this is hit then up to 10
000 points  will be won and an e
xtra life   (upto a maximum of f
ive lives)  willbe added. Howeve
r you are   only allowed three s
hots at the ship (any more are i
gnored)"
2110 PRINT : PRINT FLASH 1;"Pres
s ENTER to continue.": FLASH 0
2120 IF CODE INKEY$<>13 THEN GO
TO 2120
2130 CLS : PRINT "However if you
 miss the ship      then you lose
a life(only if youhave one life
or more in            reserve.)
                      The game conti
nues until three  invaders land
with something of interest in st
ore for those who score very hig
h."
2140 PRINT : PRINT : PRINT FLASH
 1; BRIGHT 1;"PRESS ENTER TO BEG
IN GAME!!!": FLASH 0: BRIGHT 0
2150 IF CODE INKEY$<>13 THEN GO
TO 2150
2160 RETURN
```

GAMES MACHINE

£7.95

THE FABULOUS WANDA
AND THE SECRET OF LIFE
THE UNIVERSE AND EVERYTHING
for COMMODORE 64

An adult fantasy
game with Arcade Action

You are travelling far from home looking for good times around the Universe, when you are intercepted by the evil Brutus of the Federation. Brutus forces you to go and seek the Secret of Life, the Universe and Everything - known only to the Fabulous Wanda, a hostess in the Spaced-Out Inn in Highsville on the planet COPUS. You are teleported down to Highsville where the Customs man demands money for Teleport Tax. There is a video game in the Terminal with 3 credits left.
Now you are on your own and you must decide how to proceed to Highsville Mainstreet to find the various establishments to enter and ......

OGLES for BBC/B
Designed with people of all ages in mind this programme provides an aid to learn and match colours as well as being very entertaining. Not only have colours to be matched in sequence but co-ordination skills can be developed by moving the correct coloured OGLE to match a pattern displayed on the screen. Interest and amusement are provided by you as Gordon having to control your pet dog Flash by guiding him to collect the matching OGLE, carry it back and to drop it at the correct position.
There are two levels of play.
An easy level for the younger person.
A professional level for the older person with play against the clock and Hall of Fame.
OGLE COMPETITIONS ARE FUN! £8.95

£5.95

BARRELDROP! ***for 48K SPECTRUM***

Poor Gordon! His drainpipes are blocked again and the only way to clear them is to drop barrels down them. Gordon stands on the top of his roof with 5 barrels. When the game starts he'll roll one down the roof. Press ***SPACE*** to drop it through the roof accurately into the centre of a drainpipe and you'll score the number of points in the pipe, which will start to flash, and Gordon will get the barrel back for another go. The barrel will be lost if the drop is inaccurate, or into a pipe already filled, but - Flash the dog is on hand! If you know you're going to lose the barrel and you can see Flash peeping out from the bottom-right corner, press ***D*** and Flash will save it! Once you clean all 5 pipes, you will get a bonus, but there's a surprise in store before you get the next set of pipes to fill ....

SUPERSNAILS for 16K or 48K SPECTRUM
Snails specially imported from West Africa are being kept in Dr. Van Winklehoff's laboratory for genetic experiments. The Doctor has turned them into a super-breed of snail who now leave behind a trail of super glue that will trap any snail touching it. Two of the snails, continually moving and controlled by the players, escape from their pens into the laboratory. Unfortunately, only one of the snails can escape from here into the outside world without raising the alarm. You must therefore trap your opponent and then try to escape through the small door which will then appear.
Features:- * Fast Action - 100% Machine Code
* Nine levels of difficulty
* Keyboard or Joystick Control
£5.95

£7.95

EGBERT
for COMMODORE 64
A fast Arcade Action Game for all the family to play.
Written in 100% Machine Code for super-fast action!
Egbert works on the production line at 'LEYSPACE' It was a comfortable life until the invasion of the TEBBITES from the planet TOR. Egbert's union has been exterminated and the Tebbites have left their deadly Pets running wild in the workplace. As if that wasn't enough, the evil invaders have forced Egbert to take care of an Egg - damaging the Egg will have fatal consequences for poor Egbert. Egbert is now on piecework - can he earn a decent wage? Can he even survive? **WARNING!** You may get an ulcer by playing this game.
SIX LEVELS OF SKILL.***ANYBODY WHO CAN DO LEVEL 6 SHOULD WRITE AND LET US KNOW!***

ARITHMETIC FUN-TIME for TI99
Elementary addition, subtraction, multiplication and division exercises providing valuable practice and drill for young children who are developing their basic mathematical skills.
Uses colour graphics and sounds to give a good presentation with the sums appearing in large letters on a blackboard.
The Computer plays back sums which the child has difficulty with showing the child the correct solutions. Uses the basic TI99 console. **£5.95**

Games Machine Ltd., Business & Technology Centre,
Bessemer Drive, Stevenage, Herts. SG1 2DX.
Telephone: (0438) 316561.

Please add 50p to cover post and packing.

I enclose a cheque P.O. payable to GAMES MACHINE LTD., for £ ____________

or debit my Access/Barclaycard account no. ____________

Signature ____________

NAME ____________

ADDRESS ____________

	QTY.	£
WANDA		
BARRELDROP		
EGBERT		
SUPERSNAILS		
OGLES		
ARITHMETIC		
	TOTAL	

SEND TO:-

GAMES MACHINE LTD.
Business and Technology Centre,
Bessemer Drive,
Stevenage,
Herts. SG1 2DX.

HCW34 DEALERS AND DISTRIBUTORS REQUIRED. GAMES AND EDUCATIONAL PROGRAMMES WANTED FOR ANY HOME COMPUTERS.

Andrew Thomas

PUZZLE SQUARE

Fed up with cubes? Move on to squares . . .

This program is based on the popular game in which one tries to re-arrange the numbered tiles of the puzzle to read 1-15 consecutively. Written in a fairly standard BASIC, the program should run on most machines supporting 4K or more of memory.

The computer may take up to 20 seconds to set up the puzzle; this delay is due to the set-up procedure necessary to avoid impossible puzzles. The computer will state the maximum number of moves it should take to solve the puzzle although it will sometimes be possible to complete the puzzle in fewer moves.

Replying to the question 'WHAT IS YOUR MOVE?', the player must specify first the direction of the move (Left, Right, Up or Down) and second, the number of pieces to be moved (three is the maximum). For convenience, only the first letter of the direction need be typed in and if the number of pieces to move is zero or non-existent, the computer will assume you wish to move as many as possible. Note the numbers move in the direction stated and not the space — if you find this confusing change line 250 to A$= "RLDU" as this will reverse the direction of movement.

The computer will then check to see if the puzzle has been completed and if not, print the puzzle out again (assuming the last move made was valid).

HOW IT WORKS

For those of you interested in the way the program actually works, the following might prove useful. The puzzle is

```
                    ****** SAMPLE RUN ******

+--------------------+                     +--------------------+
!                    !                     !                    !
!       1    2    3  !                     !  1    2         3  !
!                    !                     !                    !
!  5    6    7    4  !                     !  5    6    7    4  !
!                    !                     !                    !
!  9   10   11    8  !                     !  9   10   11    8  !
!                    !                     !                    !
! 13   14   15   12  !                     ! 13   14   15   12  !
!                    !                     !                    !
+--------------------+                     +--------------------+

MOVE NUMBER  16                            MOVE NUMBER  17
WHAT IS YOUR MOVE LEFT  2                  WHAT IS YOUR MOVE R1

+--------------------+                     +--------------------+
!                    !                     !                    !
!  1         2    3  !                     !  1    2    3       !
!                    !                     !                    !
!  5    6    7    4  !                     !  5    6    7    4  !
!                    !                     !                    !
!  9   10   11    8  !                     !  9   10   11    8  !
!                    !                     !                    !
! 13   14   15   12  !                     ! 13   14   15   12  !
!                    !                     !                    !
+--------------------+                     +--------------------+

MOVE NUMBER  18                            MOVE NUMBER  19
WHAT IS YOUR MOVE L                        WHAT IS YOUR MOVE U

+--------------------+
!                    !
!  1    2    3    4  !
!                    !
!  5    6    7    8  !
!                    !
!  9   10   11   12  !
!                    !
! 13   14   15       !
!                    !
+--------------------+
WELL DONE .
YOU TOOK  19 MOVES TO COMPLEATE THE PUZZLE .
THE PUZZLE COULD HAVE BEEN SOLVED IN  16 MOVES .

DO YOU WANT ANOTHER GAME PLAY AGAIN SOME TIME !!

READY.
```

stored in the 16 element array, AO, and the space is represented as a zero. P has been designated the position of the space in the array, N is the number of pieces to move and D represents the direction in which the pieces move.

Moving on to the listing itself, lines 310-450 set up the puzzle, it is printed out by lines 490-650 and the two lines, 660 and 670, check if the puzzle has been completed. The rest of the program is concerned with movement; lines 720-740 find the numeric equivalent to the direction chosen, lines 810-820 check the move vertically, lines 980-990 check the move horizontally and lines 920-960 actually move the pieces.

```
7       REM WRITTEN ON APRIL 1980 WRITTEN ON THE
        'SORCERER'
10      DIM A(16),D(4)
20      PRINTCHR$(12);TAB(26)"NUMBER PUZZLE"
30      PRINTAB(26)"[13 - ]"
40      PRINT
50      PRINT
60      PRINT"THIS PROGRAM IS A COMPUTER VERSION OF THE
        POPULAR"
70      PRINT"NUMBER PUZZLE"
80      PRINT
90      PRINT"THE OBJECT OF THE GAME IS TO REARRANGE THE
        PUZZLE"
100     PRINT"SO THAT IT READS 1-15 IN SEQUENCE, WITH 1 AT
        TOP LEFT"
110     PRINT''AND THE SPACE AT BOTTOM RIGHT."
120     PRINT
130     PRINT"WHEN ASKED 'WHAT IS YOUR MOVE' YOU INPUT
        THE DIRECTION"
140     PRINT"IN WHICH YOU WANT TO MOVE THE NUMBERS
        AND OPPISITE"
150     PRINT"DIRECTION YOU WANT TO MOVE THE SPACE."
160     PRINT"THE DIRECTIONS ARE 'RIGHT', 'LEFT', 'UP' AND
        'DOWN'."
170     PRINT" FOR CONVENIENCE ONLY THE FIRST LETTER IS
        NEEDED."
180     PRINT"BEFORE HITTING 'RETURN' INPUT ALSO THE
        NUMBER THAT YOU"
190     PRINT"WANT TO MOVE. IF YOU INPUT '0' THE PROGRAM
        WILL ASSUME"
200     PRINT"YOU WANT TO MOVE AS MANY AS POSSIBLE."
210     PRINT"THE MAXIMUM YOU CAN MOVE IN ONE GO IS '3'."
220     PRINT
230     PRINT"THE PROGRAM WILL VALIDATE YOUR MOVE."
240     PRINT
250     A$="LRUD"
260     M=0
270     D(1)=-1
280     D(2)=1
290     D(3)=-4
300     D(4)=4
310     FOR I=1 TO 15
320     A(I)=I
330     NEXT I
340     A(16)=0
350     P=16
360     R=INT(RND(1)*10)+12
370     FOR W=1 TO R
380     S=0
390     N=INT(RND(1)*3)+1
400     IF ABS(D)=4 THEN D=D(INT(RND(1)*2)+1): GOTO 420
410     D=D(INT(RND(1)*2)+3)
420     GOSUB 820
430     IF N<>0 AND S=1 THEN N=0 : S=0 : GOTO 420
440     IF N=0 AND S=1 THEN S=0 : D=-D : GOTO 420
450     NEXT W
460     PRINT
470     INPUT" HIT 'RETURN' TO CONTINUE ";Q$
480     W=0
490     PRINT
500     PRINT"HERE IS THE PUZZLE IT CAN BE SOLVED IN
        ";R;"MOVES."
510     PRINT
520     PRINT"[+][20 - ][+]"
530     PRINT"[!][20 SPC][!]"
540     M=M=+1
550     FOR I=0 TO 12 STEP 4
560     PRINT"!";
570     FOR J=1 TO 4
580     IF A(I+J)=0 THENPRINT"[4 SPC]";:P=I+J : GOTO 610
590     IF A(I+J)<10 THENPRINT"[4 SPC]";A(I+J); : GOTO 610
600     PRINT"[2 SPC]";A(I+J);
610     NEXT J
620     PRINT"!"
630     PRINT"[!][20 SPC][!]"
640     NEXT I
650     PRINT"[+][20 - ][+]"
660     FOR I=1 TO 15
670     IF A(I)=I THEN NEXT I : GOTO 1040
680     PRINT
690     PRINT"MOVE NUMBER ";M;
700     INPUT"WHAT IS YOUR MOVE";Q$
710     PRINT
720     FOR I=1 TO 4
730     IF LEFT$(Q$,1)=MID$(A$,I,1) THEN 780
740     NEXT I
750     PRINT"ENTRY FORMAT INCORRECT."
760     PRINT"NOW ";
770     GOTO 700
780     D=D(I)
790     GOSUB 810
800     GOTO 520
810     N=VAL(RIGHT$(Q$,1))
820     IFP-(D*N)>0ANDP-(D*N)<17ANDP-D>0ANDP-D<17 THEN
        870
830     IF W<>0 THEN S=1 : RETURN
840     PRINT"MOVE IS INVALID."
850     PRINT"NOW";
860     GOTO 700
870     C=1
880     IF ABS(D)=1THEN 980
890     IF N<>0 THEN 920
900     IFP-(C*D)>0ANDP-(C*D)<17THEN C=C+1 : GOTO 900
910     N=C-1
920     FOR I=1 TO N
930     A(P)=A(P-D)
940     A(P-D)=0
950     P=P-D
960     NEXT I
970     RETURN
980     E=INT((P-1)/4)*4+1
990     IF P-(N*D)<EOR P-(N*D)>E+3 OR P-D<E OR P-D>E+3
        THEN 830
1000    IF N<>0 THEN 920
1010    IF P-(C*D)>=E AND P-(C*D)<E +4 THEN C=C+1 :
        GOTO1010
1020    N=C-1
1030    GOTO 920
1040    M=M-1
1050    IF M<=RTHEN PRINT"WOW!! OUTSTANDING
        PERFORMANCE !!!!!":GOTO1090
1060    IF M<R*2 THENPRINT"WELL DONE.":GOTO1090
1070    IF M<R*4 THENPRINT"AVERAGE
        PERFORMANCE.":GOTO1090
1080    PRINT"YOU NEED MORE PRACTISE !!!!"
1090    PRINT"YOU TOOK ";M;"MOVES TO COMPLETE THE
        PUZZLE."
1100    PRINT"THE PUZZLE COULD HAVE BEEN SOLVED IN
        ";R;"MOVES."
1110    PRINT
1120    INPUT"DO YOU WANT ANOTHER GAME":Q$
1130    IF LEFT$(Q$,1)="Y"THEN 20
1140    PRINT"PLAY AGAIN SOME TIME !!"
1150    END
```

Listing 1. The program for Puzzle Square.

Derek Hufton

SERPENTS

Play this classic snake game written for the BBC Micro.

Serpents is a very simple but addictive two player game. Each player controls the movement of a snake within the screen border. These snakes can be moved up, down, left or right by the use of the following keys:

Player 1 (top)

W

A S

Z

Player 2 (bottom)

— (underscore)

] RETURN

DELETE

The basic aim is simple, do not bump into anything. You must not hit the trail. If you do, your opponent scores one point, the first player to reach ten is the winner. If the top snake crashes a high tone is sounded, if the bottom snake crashes a lower tone is sounded. In the event of a head on collision, the lower is decided randomly. At the end of a game press the space bar to play again.

Full use is made of the BBC's impressive colour graphics and sound to give a game that requires quick thinking and dexterity, and is also pleasing to watch. At the start of each game the colours are chosen randomly, so most games will be visually different, and a speed level between 1 and 5 can be selected, 1 being fastest. The listing given is for a 32K BBC machine, but for those with only 16K suggested changes are given in '**Hints On Conversion**' to enable the program to run in 16K.

HINTS ON CONVERSION

Change the following lines as indicated and the program will then run on a 16K machine:

VARIABLE FUNCTION

A%	Colour of player 1 snake
B%	Colour of player 2 snake
F%	Selected speed number
H1%	Character number of player 1 snake head
H2%	Character number of player 2 snake head
1%	X-axis position of player 1 snake
J%	Y-axis position of player 1 snake
K%	X-axis position of player 2 snake
L%	Y-axis position of player 2 snake
K1%	Increment in X-axis position of player 2 snake
L1%	Increment in Y-axis position of player 2 snake
M%	Time delay factor (hundredths of a second)

How It Runs

Line	Function
Lines 26-34	Defines the characters to be used as the snakes' head in the 4 different directions and for the body.
Lines 50-60	Defines sound envelopes, used when crashing.
Lines 80-200	Main program loop.

How It Runs

Procedure	Function
PROCmove	Lines 1020 to 1130 check to see if specific keys have been pressed at that instant and sets the appropriate user defined head character and the amount to advance on the X and Y axes. In Mode 2, one position horizontally corresponds to 64 graphics positions, one position vertically corresponds to 32 graphics positions. The keys used generate the following for INKEY: W = −34, A = −66, S = −82, Z = −98 __ = −41, ⊐ = −89, RETURN = −74, DELETE = −90 Line 1210 checks to see if the next position for player 1 is already occupied (ie if the colour is not black). Line 1220 does the same for player 2. Lines 1240, 1250 set the colour of snake 1 and moves on one position. Lines 1260, 1270 repeat the above for snake 2. Lines 1280, 1290 update the screen co-ordinates for each snake head. Lines 1300, 1310 perform the time delay depending on the chosen speed. Line 1320 flushes the keyboard buffer.
PROCborder	Lines 2015, 2017 set colour for border and clear screen. Lines 2020 to 2065 draw the border. Lines 2072, 2074 set start positions and increments for snake heads. Lines 2076, 2090 print score headings and snake heads in start positions.
PROCcolours	Lines 2100 to 2170 randomly selects the colours for the border, snakes and score headings. These will change for each game.
PROCcrash:	Line 3020 updates the score. Line 3030 generates the sound under control of the appropriate envelope depending on which snake crashed. High tone is player 1 (top snake), low tone is player 2 (bottom snake). Lines 3040 to 3050 pause until sound has played.
PROCend	Line 5004 resets the screen and prints final score. Lines 5010, 5020 selects a flashing colour to print "GAME OVER". Line 5030 waits until the Space Bar has been pressed before continuing with another game.
PROCspeed	Lines 6000 to 6040. Sets colour green, clears the screen and asks for a speed value in the range 1 to 5. Line 6030 then sets a delay factor based on the selected speed. This is used in conjunction with lines 1300 and 1310 in PROCmove. Line 6010 CHRS(7) generates a beep.

15 Change to MODE 5
2120 to 2150 Only 4 colours are available in Mode 5, one of these is normally black, so lines 2120 to 2150 become:

```
2120 C% = RND(3)
2130 A% = RND(3):IF A% =
C% GOTO 2130
2140 B% = RND(3):IF B% =
OR B% = C% GOTO2140
2150 D% = B%
```

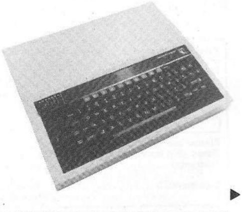
5010 GCOLO, 3 (In Mode 5 logical colour 13 is not in range)

Other Machines
To help with conversion to other micros, the following notes may be helpful:

MODE 2 This selects one of the BBC's graphics modes. Mode 3 has sixteen colours available, colours 1 to 7 can be used for game display, where:

1 = Red
2 = Green
3 = Yellow
4 = Blue
5 = Magenta
6 = Cyan
7 = White

VDU24 This sets the co-ordinates for the graphics window.
CLG Clears the graphics screen.
VDU5 Followed by the character code and eight further numbers defines a graphics character as the bit pattern of the decimal characters.
*FX15. 1 Flushes the keyboard buffer.
% The % on the end of a variable indicates on integer.



```
10   REM Serpents - D Hufton 3 Dec 1982
15   MODE2
20   VDU5
26   VDU23,231,0,62,103,255,15,255,126,60
28   VDU23,232,0,124,230,255,240,255,124,60
30   VDU23,233,40,108,238,234,250,254,126,60
32   VDU23,234,60,126,254,250,234,238,108,40
34   VDU23,241,255,255,255,255,255,255,255,255
50   ENVELOPE1,1,30,20,0,50,6,50,2,1,0,-2,100,126
60   ENVELOPE2,1,100,-30,0,50,6,50,2,1,0,-2,100,126
70   VDU24,0;0;1245;1023;
80   PROCcolours
85   PROCspeed
100  PROCborder
110  PROCmove
120  PROCcrash
130  IF S%=10 OR T%=10 THEN PROCend:GOTO 80
140  GOTO 100
200  END
1000 DEF PROCmove
1010 REM check keys and move snakes
1012 Z%=0
1015 REPEAT
1020 IF INKEY(-34) J1%=32:I1%=0:H1%=233:GOTO 1100
1030 IF INKEY(-66) I1%=-64:J1%=0:H1%=231:GOTO 1100
1040 IF INKEY(-82) I1%=64:J1%=0:H1%=232:GOTO 1100
1050 IF INKEY(-98) J1%=-32:I1%=0:H1%=234
1100 IF INKEY(-41) L1%=32:K1%=0:H2%=233:GOTO 1200
1110 IF INKEY(-89) K1%=-64:L1%=0:H2%=231:GOTO 1200
1120 IF INKEY(-74) K1%=64:L1%=0:H2%=232:GOTO 1200
1130 IF INKEY(-90) L1%=-32:K1%=0:H2%=234
1200 REM test crash and move
1210 IF POINT(I%+I1%,J%+J1%-12) <> 0 Z%=Z%+1
1220 IF POINT(K%+K1%,L%+L1%-12) <> 0 Z%=Z%+2
1230 IF Z% GOTO 1350
1240 GCOL0,A%
1250 MOVE I%,J%:PRINT CHRS(241):MOVE I%+I1%,J%+J1%:PRINT
CHRS(H1%)
1260 GCOL0,B%
1270 MOVE K%,L%:PRINT CHRS(241):MOVE K%+K1%,L%+L1%: PRINT
CHRS(H2%)
1280 I%=I%+I1%:J%=J%+J1%
1290 K%=K%+K1%:L%=L%+L1%
1300 W%=TIME
1310 REPEAT UNTIL TIME > W%+M%
1320 *FX15,1
1350 UNTIL Z%>0
1400 ENDPROC
1999 ******************************
2000 DEF PROCborder
2010 REM draw perimeter etc.
2015 GCOL0,C%
2017 CLG
2020 MOVE10,0:PLOT85,0,1023:PLOT85,10,1023
2030 MOVE5,1018:PLOT85,1245,1023:PLOT85,1245,1013
2040 MOVE1235,1013:PLOT85,1245,0:PLOT85,1235,0
2050 MOVE1235,53:PLOT85,10,0:PLOT85,10,53
2055 MOVE11,963:DRAW1234,963:PLOT85,1234,953:
DRAW11,953:PLOT85,11,963
2065 MOVE150,1012:DRAW150,964:PLOT85,1088,1012:
PLOT85,1088,964
2072 I%=22:J%=950:I1%=64:J1%=0
2074 K%=1173:L%=85:K1%=-64:L1%=0
2076 GCOL0,D%
2080 MOVE215,1000:PRINT"****SCORE****":MOVE20,1000:
PRINT;S%:MOVE1100,1000:PRINT;T%
2082 GCOL0,A%
2084 H1%=232:H2%=231
2085 MOVEI%,J%:PRINT CHRS(H1%)
2087 GCOL0,B%
2090 MOVEK%,L%:PRINT CHRS(H2%)
2098 ENDPROC
2099 ******************************
2100 DEF PROCcolours
2110 REM set colours for border & snakes
2120 C%=RND(7)
2130 A%=RND(7):IF A%=C% GOTO 2130
2140 B%=RND(7):IF B%=A% OR B%=C% GOTO 2140
2150 D%=RND(7):IF D%=A% OR D%=B% OR D%=C% GOTO 2150
2170 ENDPROC

3000 DEF PROCcrash
3010 REM crash routine
3015 IF Z%=3 THEN Z%=RND(2)
3020 IF Z%=2 THEN S%=S%+1 ELSE T%=T%+1
3030 SOUND1,Z%,53,40
3040 W%=TIME
3050 REPEAT UNTIL TIME>W%+150
3100 ENDPROC

5000 DEF PROCend
5004 PROCborder
5010 GCOL0,13
5020 MOVE350,650:PRINT"GAME OVER"
5030 REPEAT:I%=GET:UNTIL I%=32
5050 ENDPROC

6000 DEF PROCspeed
6002 GCOL0,2
6004 REPEAT
6006 CLS
6010 PRINTTAB(2,4);"SPEED (1 - 5) ";CHRS(7);:INPUT F%
6020 UNTIL F%>0 OR F%<6
6030 M%=(F%-1)*5
6035 S%=0:T%=0
6040 ENDPROC
```

DEMON KNIGHT

A terrifyingly difficult adventure.

Only £6.50 (plus 25p p & p)
Available for: Spectrum, BBC B and Commodore 64

Had any good Adventures lately?

Try this one. It is difficult, deadly and logical. Your task is to find, face and defeat the Demon Beelzebub. You will need a strong nerve and a clear, incisive mind to succeed.

In order to defeat the Demon, magic must be used – so don't think you're just going to stroll up to Beelzebub and start swinging a sword around. Not a good idea. This is an Adventure in which it will be a long time before you get **that** far – so don't blow it!

You only get one life, but you can save onto tape at any stage and restart later if it all gets too much to cope with. DEMON KNIGHT is a challenge to Adventurers of all levels – can you meet it?

Fill in the coupon and return it to:
ASP Software
145 Charing Cross Road
London WC2H OEE.

Please send me . . . . tape(s) – (delete as necessary) of Demon Knight for. . . . . (state which version required). I enclose my cheque/ Postal Order/ International Money Order (delete as necessary) for: . . . . (made payable to ASP Ltd) OR Debit my Access/ Barclaycard (delete as necessary)

☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐

Please use BLOCK CAPITALS
Name (Mr/ Mrs/ Miss). . . . .
Address. . . . .
. . . .
Signature. . . . . Date. . . . .
Please allow 28 days for delivery

Simon Goodwin

INTER—PRETERS

Just what does turn all that BASIC code into something the computer can run? The answer is a special program called an Interpreter.

Almost without exception the current range of personal computers use an 'interpreted' BASIC as their main language. This means that when you enter a normal BASIC program you are not loading the computer memory with the machine code instructions your processor will execute when you type RUN. Instead you are loading a stream of symbols which describe what you want to do to a large machine code program called the 'interpreter', this then executes many thousands of instructions trying to work out what you want to do. An interpreter looks at a set of 'rough notes' (rather like shorthand) and then uses a kind of programmed initiative to work out what the symbols mean and to carry out the appropriate operations.

This process is fundamental to an understanding of how a micro-based system works: this article will explain (in general terms) how your interpreter executes a BASIC program. We'll also compare the common interpreters with 'compilers', explaining the difference between the two. Whether your computer is an Apple, PET, TRS-80, Sharp, NASCOM or whatever, the principles used are much the same.

INTERPRETATIONS

Take this simple program line as an example:

```
100 A = B+1: GOTO 200
```

We'll assume that it is a single line from a larger program, and that a while ago you typed RUN and the computer has just reached the line above. It looks at the first part of the line, after the line number, and scans through it until it finds a character that is not alphanumeric — in this case, the equals sign. It then looks at the character it has read so far (the letter A) and checks that it is not a reserved word (LET, IF, GOTO, etc). In this case it decides that 'A' is not reserved. Consequently the computer assumes that the statement is an assignment. If your BASIC requires you to use 'LET' statements then it would give an error message at this point, but most popular interpreters permit LET to be optional.

The interpreter assumes that 'A' is a variable name. A routine is called to search a table of current variables. This is set up as the program runs and comprises a list of variable names and their values. The routine will search through for the name 'A'. When it finds the entry it will store its memory address in an area of RAM reserved for the use of the interpreter (either on the stack or in a fixed place). The main section reads further and finds the name 'B'. It calls up the variable-search routine again which tries to find 'B' in its table. If it can't find it, the routine creates a new entry at the end of the table enabling it to find that variable the next time it is needed. Most versions of BASIC (but not that on an Acorn ATOM) will set the value of the variable to zero when it is created.

The search routine is finally able to return the address of the variable B to the main interpreter. The main program can now look at the fourth character on the program line! It finds the plus sign (which also marked the end of the name B) and makes a note that it will have to do some adding — but first it must work out what to add. It reads further and locate the figure '1' then the colon which signifies the end of the statement. Other types of BASIC might use a different character to mark the end of a statement — on a Sinclair computer, for example, the end of the line signifies the end of a statement since only single-statement lines are allowed.

Now the computer calls a routine to convert the figure '1' from its format in the BASIC program (an ASCII character with code 49) into the binary format used by the microprocessor. When you type in your numbers in base ten using one character for each figure. When your computer does arithmetic in base two it uses one character for every eight figures! Consequently the interpreter has to use a quite complicated routine to convert numbers from decimal to binary. Even though in this case we are only converting a single-digit number (in either base!), the routine used must work for all lengths of number allowed by the system and make the appropriate changes if it finds a minus sign or a decimal point. Once it has discovered the correct value, it can call a relatively short subroutine to actually add what it has worked out to the number in the workspace area. Once the calculation is complete, it retrieves the location of variable A (remember variable A?) and stores the result there.

Hardly pausing to catch its breath, the interpreter now has a look at the next statement. It checks that GOTO is a valid word — this is where your WHAT. . .IF and IF. . . NECESSARY statements get thrown out. In the case of this line, the GOTO part tells the interpreter to get ready to change line numbers. It reads the next part of the statement — the line number — and converts it into a binary number. Then it will merrily search all through the program lines in the ▶

memory trying to find a line with a number that matches the one in its buffer. If it finds one, it stops searching and gets ready to work out or 'parse' the first statement on the new line. If it fails to find a match anywhere in the program, it gives up and prints an error message; unlike the case of variables, it is not considered good form for an interpreter to create a new line of program if it can't find the one requested! The only computer which comes close to doing this is the ZX81 which jumps to the line with a number closely following the one chosen if you try to GOTO a line which isn't there.

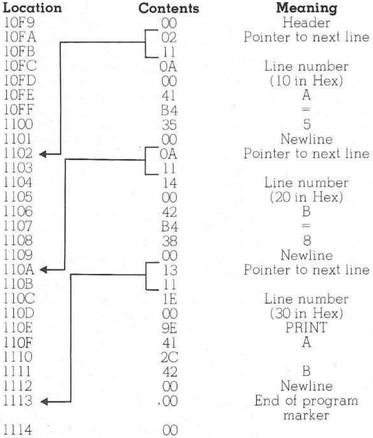
TOO SLOW?

That was a simple example of the execution of a program line but it summarises the workings of virtually all BASIC interpreters. Various programming tricks are somtimes used by their authors to make the program text easier for the interpreter to parse or to convert; reserved words may be compressed into special character-codes, or pointers may be used to help the interpreter find its way from one line to another. Whatever methods are used, a relatively small proportion of the time during the running of an interpreted program is taken up in searching through tables in memory or in converting information from one format to another.

All this may not worry you much as a personal computer user. So long as the machine can do it faster than you can work it out on your fingers or at the typewriter, all is well, Sooner or later, however, you will find that your program isn't running quite as fast as you would like. Maybe it is the unnerving way the Space Intruders judder to a halt when you try to move your laser and shoot them at the same time – maybe it's the long wait while your computer sorts your massive list of friends into order of protocol. Either way the fact that the computer is executing a few thousand machine instructions evey second doesn't seem to help much. You will probably have come across a variety of suggestions to be used to speed up programs – peculiar things like using as few variables as possible, putting your subroutines at the start of the program, using variables instead of constants, etc. Maybe this article will explain how some of these tips are of use in reducing the overheads imposed by the interpreter.

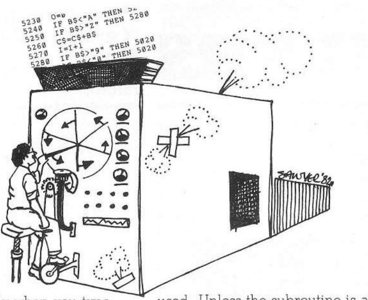
Another chronic problem when a computer uses an interpreter is the way it never learns by its mistakes. You would expect any sensible machine, surely, to know where line 200 is after it has been sent there a few times? Unfortunately this is not the case; each time the interpreter executes a line it has to start working out what the contents mean from scratch.

Hopefully you have now guessed what a compiler does. It translates a program written in 'source' form – the PRINT and INPUT statements you are used to writing – into a program which the processor can execute more or less directly, without all that searching and conversion. A compiler usually makes two or more 'passes' or repeated searches through the program text from start to finish. Usually the computer must be told by a special command to compile the program before it can be RUN at all. There is a vital difference here between compiled and interpreted programs: when you type RUN under an interpreter, you are telling the 'editor' (a part of the BASIC which allows you to type in lines) to start executing the interpreter; when you type RUN under a compiler, you are telling the computer to actually execute the machine-code generated when the source was compiled. A few compilers will operate



Location	Contents	Meaning
10F9	00	Header
10FA	02	Pointer to next line
10FB	11	
10FC	0A	Line number
10FD	00	(10 in Hex)
10FE	41	A
10FF	B4	=
1100	35	5
1101	00	Newline
1102	0A	Pointer to next line
1103	11	
1104	14	Line number
1105	00	(20 in Hex)
1106	42	B
1107	B4	=
1108	38	8
1109	00	Newline
110A	13	Pointer to next line
110B	11	
110C	1E	Line number
110D	00	(30 in Hex)
110E	9E	PRINT
110F	41	A
1110	2C	
1111	42	B
1112	00	Newline
1113	.00	End of program marker
1114	00	

Fig. 1. How a short section of BASIC is stored in the memory. The program reads from top to bottom and the BASIC has been tokenised.



automatically when you type RUN – they compile and execute as one step, as far as the user can tell. Some will compile each line as it is typed in although these are not usually as efficient a the ones which treat compilation as a separate process.

Generally a compiler will first pass through the program text working out where each line will be in the final code and reserving space for the variables used. Once that analytical pass is complete, the compiler is able to go back through the BASIC program, converting GOTOs and similar instructions into direct machine code jumps to the lines concerned, having worked out the appropriate addresses earlier.

All the variable references are converted into simple instructions which tell the processor to transfer information to and from memory. Constants (eg 32767, 0.778, 1.6775E-6, "TOAD") are converted from the text typed by the programmer into binary that can be easily used by the micro. It is usually still necessary to provide some routines outside the actual program code – for example, most microprocessors cannot handle floating-point calculations internally – so these functions are set up as subroutines and either built into a 'library' at one end of the compiled program or inserted into the code when they are first used. Unless the subroutine is a very simple one, it will only be built in once. Other parts of the program that need it will just call it when necessary by loading memory or registers with the parameters (the data to be processed and a note on where to put the result). Then it calls up the required routine. The final result is a machine code program which will do everything that the equivalent BASIC would have done – but maybe 10 times faster!

THE REASONS WHY

At this point it is worth considering the advantages of using an interpreter. The most important advantage becomes apparent when you try to modify a compiled program . . . you will usually have to reload and recompile the entire source listing source listing before you can test even a one-line change. Once a program has been compiled the original text is no longer in the computer memory and the machine code that replaces it is almost impossible to modify.

In the compiled program you cannot simply insert a few extra instructions; the rest of the program relies upon the fact that the instructions (variables and so forth) are in fixed places, worked out when you first compiled. After you have changed a compiled program those addresses will probably be wrong. A jump which used to send you to line 100 of the original program might now drop you into the middle of the previous line. The source program must be translated again before it will work. An interpreter makes a search for each item of data whenever it is referred to, so the problem doesn't arise.

This problem becomes even more annoying when you are trying to track down an elusive bug in a compiled program. An interpreter will let you insert STOP and PRINT statements at crucial places in the program; you can even patch in a GOTO to skip over a given section of the listing. It is easy to start a program from a point in the middle, perhaps using variables set up earlier but as most compilers generate pure machine code (without line numbers or comments), they rarely permit interruptions of this kind and each and every temporary change must be compiled with the whole program. As a compilation usually takes minutes rather than seconds, the speed improvement offered by a compiler begins to look slightly less attractive to the programmer.

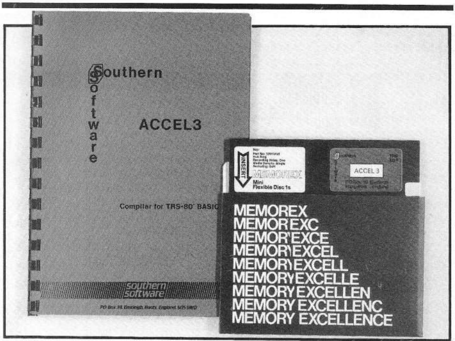
WHICH TO CHOOSE?

So far we have made no mention of the relative sizes of compiled and interpreted programs. The main advantage of a compiler is that once it has done its job, it can be removed from memory as the compiled program is self-supporting machine code. An interpreter must be resident with your program when you type RUN (really the ***interpreter*** does the RUNnning!). In practice quite a few routines used by an interpreter must be duplicated inside a compiled program – for example, code to handle arithmetic, read from the keyboard or write to the screen. Generally a compiled program will take up slightly less space than an interpreted one with its interpreter but the difference is not great. Many micros have their interpreters in ROM so that the space occupied cannot be used by a compiler anyway.

Simon Goodwin

COMPILERS

Here is another method of getting BASIC programs into a form that can be used by the micro.



In the article on Interpreters we explained some of the features of compilers for BASIC personal computers. Here we look at one ingenious commercial product — the ACCEL2 compiler for TRS-80 and Video Genie. This program demonstrates may of the strengths and weaknesses of compilers in general. We're grateful to Southern software, the UK authors, for permission to describe some of its inner workings as they illustrate many of the points that a user should consider when attempting to enhance the features of almost any BASIC micro. There is no reason why a compiler along the lines of ACCEL2 could not be written for any of the popular computers.

LOADING AND USING ACCEL2

Briefly, a compiler is a program that translates BASIC statements from the 'tokens' in memory (eg FOR, GOTO, etc) into machine code. Once a program has been translated it is much more difficult to modify than when it was 'interpreted' but it can be run ten or more times faster!

The compiler was supplied on digital cassette, accompanied by a strikingly-bound 18-page manual (it is compatible wth disc systems). Two identical copies of the program are recorded on the tape, which is loaded using the Level 2 BASIC SYSTEM command since it is a machine-code routine. When first run, the message "TARGET ADDR?" appears. This feature allows you to store the compiler anywhere in your computer's RAM. The program will automatically move itself to the target address that you specify in answer to the prompt. That means it is compatible with all 'memory sizes' (so long as there is enough room for it to load in the first place). Normally you should reserve an area of memory out of reach of BASIC when the machine is switched on. The program doesn't check that the memory address specified contains free memory but only the most confused programmer would consider that a problem. The manual contains a useful table of suggested storage areas for different systems.

The ACCEL2 compiler is remarkably small — it only occupies 5½K of user RAM and loads in less than two minutes. About three-quarters of it can be deleted once compilation has taken place (most of the program does the translating rather than suport the operation of the final compiled BASIC program). Once the compiler is in memory, you may load or type in any BASIC program that will fit into your remaining RAM. You may even RUN the program in the normal way and it will execute as if the compiler was not there. When you think there are no more errors in the program, you may compile it by typing the new command /FIX. A 14K program should compile in about two minutes. There is no way to interrupt the compiler while it is working; if you press the Reset button before the prompt re-appears, you will have to reload both programs before you can get any sense out of your system.

HOW IT WORKS

The compiler works by looking through the BASIC program in memory and trying to find instructions that could easily be performed in machine code. It would not be possible to fit a full-feature compiler into only 5632 bytes of Z80 machine code, so ACCEL2 sets out to translate only those functions that can be enhanced most dramatically by compilation. It relies upon the fact that once the program is compiled, it cannot be edited. That means that most of the searches carried out by the interpreter can be replaced by simple instructions that fetch data directly from some pre-calculated memory address. These addreses are worked out by ACCEL2 while compilation takes place and are stored immediately inside the compiled program. The Z80 processor used in Video Genie can only directly process numbers in the range −32768/+32767 (the range that can be stored in one of its 16 bit registers). Integer (whole number) values on most mini and micro-computers fall into this range and are useful as general-purpose stores in programs. Loop counters, graphics co-ordinates and 'flags' are generally integer values and Apple, Tandy and PET computers all support special 'integer variables'.

The Z80 instruction set includes integer addition and subtraction but not multiplication, division or complex functions such as square-root. These must be performed using a large number of machine code instructions and consequently ACCEL2 does not directly compile them. All GOTOs and other line number branches can be compiled since the target of a branch is static. ACCEL2 replaces those instructions by machine code JUMPs. This is a most valuable improvement over interpreted BASIC – a Z80 jump takes about five millionths of a second on a TRS-80 or Video Genie (even less on a computer with a 6502 processor like Apple or PET). The BASIC interpreter won't even have worked out what comes after the GOTO, let alone check the syntax, convert the number to binary and search all the way through memory to find the line required!

There are quite a few other BASIC functions that can be handled directly by the processor. Graphics commands such as SET, RESET and POINT are really only elaborate ways of setting or clearing bits in the computer's video RAM. ACCEL2 replaces them by calls to a set of simple subroutines. GOSUB and RETURN have direct machine code equivalents, and very little code is needed to simulate a FOR . . . NEXT loop since it usually only uses integer addition, comparison and jumps (so long as the index variable is an integer – and this is usually the case). PEEK and POKE are simply compiled and constants (such as "FRED", –1000, 3.14, etc) are stored as binary values.

As well as translating those statements into machine code, ACCEL2 is faced with a large number of other more complex operations: floating-point calculations; string-handling; and so forth. These are compiled in a rather different way. The data to be processed is fetched as detailed above but rather than operate upon it directly, ACCEL2 calls up routines within the ROM interpreter to actually work out the results. Even though it is the same code 'doing the work', the BASIC intepreter normally spends so much time searching for data that ACCEL2 can usually speed up those functions by three or four times. Most functions that are very complex (such as SIN and all the varieties of PRINT) are left in their interpreted form, keeping the size and complexity of the compiler down. You may choose to compile only a part of your program at a time and ACCEL2 will automatically take over from the interpreter when it comes to a compiled statement, relinquishing control later. It is this feature that makes ACCEL2 interesting to the programmer since it permits a compiler to be written piecemeal, one function at a time. In fact ACCEL2 was written in this way – the original 'ACCEL' being a 2¾K program that only compiled branches and most integer operations. It is still available (for half the price of ACCEL2) and is ideal for games and similar programs.

ACCEL2 converts compiled statements into REMs followed by machine code. To prevent confusion and to reduce the size of the compiled program, genuine comments are removed from the BASIC before it is compiled. Despite this, compiled programs are usually larger than their intepreted equivalents.

HANDING OVER

To accomplish the handover of control between compiler and interpreter, a very valuable feature of Microsoft (and most other) BASIC interpreters is used. You may have been disappointed to discover that on your PET, TRS-80 or Apple not all of the internal memory can be used to store BASIC programs. The interpreter borrows a few hundred bytes to store partial results of calculations and various other notes on the progress of a session at the keyboard. There is also a table of memory addresses (or JUMP instructions) in RAM and the interpreter calls at those locations at various key points in the execution of a program. There are about fifty 'vectors' in reserved memory on a Video Genie or TRS-80, and two of them are of special interest to compiler authors since they ar used just before each direct command is processed and before statements are executed. On a Video Genie or TRS-80 (Model 1 or 3) these vectors are stored at 41B2 Hex and 41C4 Hex – other interpreters will have equivalent vectors at different addresses.

Normally the vector locations contain nothing more than a jump back into the depths of the BASIC ROM but if a machine code program wants to take over at times, it need merely replace the vector in RAM with one pointing to its own routines. It can check what's going on when it is called up and either jump back to ROM if it doesn't want to interfere or process the statement in its own way and then re-enter the interpreter at the point where the next instruction is fetched. Extended BASICs and 'Toolkits' use these vectors to add to the commands on a computer – once the relevant locations have been found, a user can expand the system software of his computer little by little; there are books available describing the workings of most of the popular microcomputer interpreters. My recommendation for TRS-80/Video Genie users is 'Pathways through the ROM' distributed by The Softwarehouse.

To ensure that ACCEL2 will use the same variable storage area as BASIC, it checks through memory during compilation working out where

▶

each variable will be stored and where each line of the program will eventually end up. To make the compiled program exactly compatible with normal BASIC, ACCEL2 has to do a certain amount of 'housekeeping' as it runs. For this reason the compiled program is usually somewhat larger than the original version. To try to minimise that effect, ACCEL2 removes comments and unnecessary spaces form the BASIC before compiling it. ACCEL2 will issue an OUT OF MEMORY error if the compiled program ends up too large to fit in your computer. Often you can get around this by only compiling the part of the program that is executed most often. This keeps code expansion to a minimum and with care, the mixed code will run almost as fast as if it were all compiled. After such an error you have to reload a copy of the original 'source' BASIC entered using the interpreter; you cannot edit the partially-compiled program to cut it down.

ACCEL2 has facilities to allow a compiled program to load another from disc. It automatically looks for a compiled version first and then for an interpreted one if the first search fails. That feature allows a disc user to compile a set of linked programs one at a time (without having to re-compile at each stage). The new disc commands are /LOAD, /SAVE and /RUN.

EXAMPLE PROGRAM

Figure 1 shows a simple program before and after compilation. The '%' signs indicate that X and Y are integer variables. These should be used whenever possible in compiled programs since they can be processed much faster than normal floating-point variables. It is not very valuable to try to compare the speed of individual instructions when interpreted and compiled, since the larger the program the longer the interpreter spends finding a given variable or line. A compiler will take a long time to translate such a program as it is doing at one step all the searching and converting that the interpreter does bit by bit (and generally over and over again!). This tiny example was compiled by ACCEL2 in less than a second and ended up using rather more memory than the original — 169 bytes instead of 126. The routine simply turns the entire computer display white by individually turning on each of 6144 graphics pixels. Interpreted BASIC took 36.5 seconds to execute the program; after compilation it ran in just five seconds. This speed improvement of around seven times is obviously not typical since most programs will use complex functions such as PRINT and decimal arithmetic. There again, ACCEL2 is at its least impressive when compiling small programs, and most programs will be accelerated by at least a factor of two or three times if they are compiled.

```
100  REM ** ACCEL2 DEMO PROGRAM
110  FOR X%=0 TO 127
120  FOR Y%=0 TO 47
130  SET (X%,Y%)
140  NEXT Y%
150  NEXT X%
160  END
(126 BYTES, RUNS IN 36.5 SECONDS)
100
110  REM
120  REM
130  REM
140  REM
150  REM
160  END
(169 BYTES, RUNS IN 5.0 SECONDS)
```

Fig.1. The example program before and after compilation by ACCEL 2.

The compiled program may look rather odd; the original REM has been removed from line 100 but the line number alone is left there in case it is referenced by other parts of the program. The REMs conceal the compiled machine codes; ACCEL2 deliberately prevents the machine code from being listed (it wouldn't make sense, either to you or to the LIST routine . . .). END is never compiled so it appears unchanged.

COMPILER RESTRICTIONS

ACCEL2 cannot compile array references with more than one dimension but this is not normally a major restriction. The size of an array must be known at compilation time (10 INPUT A 20 DIM CD(A) is not legal) but this can be avoided by always dimensioning arrays to the largest size required. ACCEL2 will be of little use in programs limited by the speed of peripherals such as disc, tape or printer. A version of Conway's game of LIFE has been written and found to be 39 times faster under ACCEL (the original 2¾K subset of ACCEL2) than when the BASIC interpreter alone was used. However, the program was written especially to be compiled and consequently a slightly faster interpreted routine could probably be found.

READ and DATA statements are not compiled since ACCEL2 cannot tell whether DATA is going to be stored in string or numeric format when the program is run. READ and DATA can be invariably be replaced by assignment statements so this is not a major problem. ACCEL2 will only compile statements that it can translate completely. Although SIN, TAN and most other complex arithmetic operations are not compiled directly, ACCEL2 uses a neat dodge to compile comparisons using them. The line:

```
300  IF SIN(X)=SQR(X) THEN 100
     ELSE 200
```

is compiled into:

```
300  TO%=SIN(X)=SQR(X):REM
```

(That first statement is perfectly legal interpreted BASIC!) The compiler uses TO% as a temporary variale to store the result of the comparison (try PRINT 2=2 and PRINT 2=1 on your computer). The compiled code behind the REM tets the value of TO%; it will be zero if the values are not equal, −1 if they are. TO% is not a legal user variable anyway because the BASIC editor treats it as a reference to the reserved word TO.

Most compilers generate a good many problems as well as advantages by comparison with an interpreter. ACCEL2 is a special case since it remains dependent upon the computer's ROM interpreter. It is possible to test a program 'in slow motion' with the interpreter and the usual BASIC debugging aids, and then to compile the program when it is (hopefully) more or less free of errors. This is important since there is not much room for error-trapping code inthe 1280 bytes of run-time routines. After ACCEL2 has been used you cannot edit the program, delete lines or change the text in any way. If you try to do so, the interpreter will 'fall over' the machine code generated by ACCEL2 and the system will probably re-boot. You may still set or print variable values in immediate mode and then GOTO the start or middle of the program to test specific routines. It can be risky to GOTO lines in the middle of the program since ACCEL2 will not recognise an accidental 'Return without Gosub' error and will probably jump away to some indeterminate location quite possibly crashing in the process. GOSUB and REM are not allowed as immmediate commands since ACCEL2 uses a different type of GOSUB from BASIC and it uses REM to signify compiled code. When the machine holds a compiled program, you should not use the commands EDIT, AUTO, CLOAD?, CSAVE, DELETE, MERGE and SAVE, since they assume that the program is in interpreted format.

The compiler gains some of its extra speed by dispensing with part of the 'housekeeping' done by the interpreter; it doesn't check the current line number while executing compiled code so ON ERROR GOTO may not go where you want it to! Likewise TRACE will only display the line numbers of statements that have not been compiled. ACCEL2 doesn't check the 'Break' key while it executes compiled lines; if you 'get stuck' in a compiled loop you will have to use Reset to get out.

We were only able to find one minor bug in the purchased compiler; if INKEY$ was followed by certain statements a cumulative Out of Memory error could develop. However, this problem has been fixed in the current issue of the compiler.

ACCEL2 imposes a number of subtle restrictions upon the programmer. Lazy BASIC programmers have been known to write code that jumps out of a loop without terminating it, as in Fig. 2. This may fail under ACCEL2 since the ocmpiler never realises that the loop has been terminated. The code in Fig. 3 works correctly whether interpreted or compiled and as BASIC on the Apple 2 also requires this construct, the limitation is quite a reasonable one. The program will also fail if the default type of variable is changed (eg from integer to string). This can sometimes happen by accident when Microsoft 12K BASIC is being used since variables are assumed to be floating-point until declared otherwise. Arrays should be dimensioned at the start of our program.

```
10  REM ** ACCEL2 COMPILER
    RESTRICTION
20  GOSUB 50
30  PRINT "RETURNED"
40  STOP
50  FOR I=1 TO 10
60  IF I>X THEN RETURN
70  NEXT I
80  RETURN
```

Fig.2. A lazy programmer's loop which might fail under compilation.

```
10  REM ** ACCEL2 REVISED LOOP
    CODING
20  GOSUB 50
30  PRINT "RETURNED"
40  STOP
50  FOR I=1 TO 10
60  IF I>X THEN J=I:I=10:REM ** SAVE
    VALUE OF I BEFORE RETURN
70  NEXT I
80  RETURN
```

Fig.3. The correct way to do it for both interpreted and compiled programs.

With the exception of the INKEY$ bus, these restrictions are clearly listed in the ACCEL2 manual. Generally the compiler will work faultlessly on programs that have been written with it in mind and the effort needed to conform with its idiosyncracies is not great. A machine code monitor is required to save compiled programs on tape since they are a mixture of BASIC and machine code, and must be saved along with the 1280 byte ACCEL2 run-time routines. If this is done a compiled program becomes a self-contained file that can be loaded from SYSTEM and then RUN as if it were high-speed BASIC. If no monitor is available then all of ACCEL2 must be loaded whenever a compiled program is required and each program has to be re-compiled from the source before use.

CONCLUSIONS

ACCEL2 is a British development that illustrates a fascinating system of 'selective compilation'. The same techniques could be applied to almost any BASIC micro-computer and in fact, an even more powerful compiler could be developed step by step. Program compilation seems certain to become a popular technique in the future — ACCEL2 demonstrates an ingenious approach that combines many of the best points of compilers and interpreters.

Southern
Software
ACCEL3
TRS 80®
P.O. Box 39 Eastleigh, Hampshire, England

The cassette version of Accel is totally relocatable, a useful facility allowing you to have a variety of other programming or editing aids resident in memory at the same time.

POSTSCRIPT

Since ACCEL2 was reviewed we have received a pre-release copy of the latest Southern Software compiler, imaginatively named ACCEL3. The new program is apparently a complete re-write of ACCEL2 and incorporates some new features.

ACCEL3 will compile non-structured FOR. . .NEXT loops making it possible to compile programs containing jumps out of loops, conditional NEXT statements and so forth. ACCEL2 didn't do this often leading to changes being made to a program before it could be compiled. The snag is that the extra code to handle unstructured loops slows up compiled programs — FOR. . . NEXT statements used with the new compiler are about half as fast as they were under ACCEL2. Similarly ACCEL3 now compiles references to arrys with more than one dimension but the speed of access to one-dimensional arrays has suffered.

ACCEL3 compiles some functions that ACCEL2 used to leave for the BASIC interpreter to handle. In particular, floating-point FOR. . .NEXT loops and functions such as INT and SQR are now compiled into ROM calls. The compilation of the functions (eg SIN, etc) doesn't really speed them up since they take much longer to process than to interpret but it does mean that expressions using them can be compiled. This would speed up the multiplication in X=SIN(X)*3.1416, for example. The USR(n) function, used to call a machine-code routine, is no longer compiled. ACCEL3 will also compile programs which use variable-bound arrays, such as — 20 INPUT N : DIM A$(N,2).

The ACCEL3 sales literature claims it is faster and generates more compact code than ACCEL2, but the difference in performance does not seem to be that great; the new compiler no longer has the disc commands /SAVE, /RUN and /LOAD. ACCEL3 allows compiled programs to be SAVEd, RUN and LOADed just as if they were normal BASIC although they will not work unless the run-time routines of ACCEL3 are in memory. Even the cassette commands, CSAVE and CLOAD, can now be used to store and retrieve compiled programs.

The pre-release version of the ACCEL3 compiler has been tested using it to speed up a few well-known programs — it even found one or two unnoticed syntax errors! More than half of the programs compiled first time and most of the rest could be compiled once a few lines were shortened or expressions simplified.

EDITOR'S NOTE

Since this article was originally written, ACCEL2 is no longer available (ACCEL3 is) but it still serves as a good example of a compiler.

GUIDED DISCOVERY
from
ETNA SOFTWARE
Have the children finished playing?
Time they started learning? They've done Tables tests and Hangman?
WHY NOT TEACH THEM ABOUT THE BBC MICRO?
GUIDED DISCOVERY is a suite of ten programs designed to stimulate an interest in HOW programs work. Aimed at age 9+, every program is simple yet effective in structure.
The cassette comes with approximately 60 pages of guidance - personalised with the child's name if you wish.
COVERS THE FOLLOWING TOPICS:
Sound, *Keys, Animation, Graphics, Filing, Time, Screen Plotting, Loops, Modes, RND, etc.
★ FULLY LISTABLE ★ PARENTS' NOTES
★ EASILY FOLLOWED ★ WELL REMD
★ EDUCATIONAL ORDERS WELCOME
To receive your copy send £9.95 + 80p p & p to:
ETNA SOFTWARE, WEST END HOUSE, WEST END LANE, MARSHCHAPEL, LINCS.
Please include your name and address and your child's name IF you wish the written material personalised.

ALLIGATA • SIMON HESSEL • ULTIMATE •
IMAGINE • PSS • MICRODEAL • SALAMANDER •
BBC • DURELL • SUPERIOR •
MORRISON MICROS • ACORNSOFT • IJK • QUICKSILVA •

Business Games for the BBC
Database for the Dragon
Ultra for the Oric
Splat! for the Spectrum
and BOOKS GALORE

Over 550 different titles available off the shelf from

THE DATA STORE
6 Chatterton Road,
Bromley, Kent.

TEL: 01 - 460 8991

Mike James

ELEGANT PROGRAMMING – 1

We start this series by explaining the difference between programming and coding.

TRS-80 ASSEMBLY LANGUAGE MADE SIMPLE
Pocket Guide to Programming
APPLE II

If you've just finished learning your first computer language, you might be worried by the fact that even though you are convinced that you understand the language it is still very difficult to write programs. However, this should not come as any surprise because knowing a language and having anything sensible to say in it are two very different things! In other words:

Knowing a computer (or any other) language does not mean that you know how to* use *it.

The question which is immediately raised is 'how do you become a programmer?' The most obvious answer is that it is only experience and practice that changes a novice into a skilled programmer. It is the purpose of this series of articles to explain some general programming methods and ideas and so shorten the time it takes for a beginner to graduate to an expert. However, this is not to say that the expert will find nothing of interest!

The techniques to be discussed in these articles are not dependent on any particular computer language. However, to make it possible to give examples and illustrations, it is necessary to use a particular language and because BASIC is such a popular language there is an obvious advantage in using it. So, to be able to understand the examples, etc, you do need to be able to read BASIC.

Each part of the series will be as self-contained as possible but it will sometimes be necessary to make use of material introduced earlier. In this first part of the series we will look at what a program is and what methods can be used to help in its production.

SOLUTIONS AND PROGRAMS

A program doesn't really have anything to do with a computer! People were using programs long before the invention of the digital computer, for example, recipes, knitting patterns and mathematical formulae are all examples of programs. A program is nothing more than a list of instructions leading to some predefined end — a meal, a scarf or the solution to a quadratic equation. It is unfortunate then that 'program' has come to mean something especially to do with digital computers as this causes the emphasis to be placed on 'computer' rather than 'program'. So 'to program' has come to mean the mastery of a computer language along with all its particular grammar. An expert programmer is supposed to be one capable of using many computer languages. Whereas, in fact, a programmer may know many languages and know nothing of programming!

The ability to program is in part the ability to find *solutions* — people who are good at solving problems usually turn out to be good programmers; those not too good at problem solving take a lot longer and some never make it at all! Problem solving in general can be taught and if you're not too good at it then don't give up — it is possible that no-one ever showed you how to tackle a problem.

Solving a problem gives you the solution.
Knowing how you solved the problem gives you a program.

CODERS

To make things clear let's examine which stage in the production of a computer program is correctly called programming. First, some ▶

statement of the problem must be found. However, this is not programming. Second, a few basic requirements for achieving the solution are outlined. This is connected with programming but is more about what computers and humans can do. Third, a sequence of steps leading to the solution is proposed. This *is* programming. Fourth, the program is realised as a written program in a computer language. This is the stage that is most often referred to as programming. It is, in fact, the least skillful of all the stages and involves mainly the correct placement of commas and other matters of simple grammar. Given a program in the form of a detailed explanation or flow diagram, the translation to code can be done automatically, and if this were programming, it would be a very dull subject indeed. In short,

CODING is not PROGRAMMING.

EXPRESSING THOUGHT — ALGORITHMS

All that we have said about programming is obvious from a consideration of normal human behaviour. Just as a thought is independent of the language used to express it — the colour red is the same concept whether written in English or German — a program is independent of the computer language used to express it. The fact that it is possible to convert a program from one computer language to another should convince even the most practical mind that something abstract lies behind any program: they both use the same ***algorithm.*** In simple terms, an algorithm is a way of doing things and it can be expressed in many ways.

At this point it is important to realise that although it is convenient to think of an algorithm as something separate from a computer language, it is impossible to express an algorithm without using language. It is often thought that the flow diagram is in some way a 'pure' expression of an algorithm but it is certainly no better than expressing it in language. Indeed, it's much ***less*** useful as no one has produced a computer that reads flow diagrams (yet)!

FINDING AN ALGORITHM

So far it sounds as though producing a program is a very magical process. You read the problem, go off into a dark corner and an algorithm enters your head from nowhere and the rest is just coding! This is, of course, nonsense! Splitting the production of a program into finding an algorithm and coding doesn't make it any easier but it does help to identify ***where*** any difficulties lie. An algorithm is similar to an English sentence — it has a verb, telling us what to do, and a noun, the object that we do it to! For example, in BASIC the instruction:

```
B+C
```

tells us to add (verb) the contents of B (noun) to the contents of C (noun). In programming languages the nouns are usually called *DATA* and the verbs are given a wide variety of names including — operators, functions, executable statements, etc . . . There is also another type of instruction that we might find in an algorithm as sometimes it is necessary to define what a 'noun' or 'data' object is. For example, in BASIC the statement:

```
DIM B(20)
```

contains a noun 'B(20)' but no verb! What it is doing is describing the object — ie it is an array of 20 elements.

Computer languages vary in the amount of data definition they require for a program. The language Pascal requires ***every*** data object to be defined before its use, but BASIC is a little more forgiving and supplies a wide range of predefined types such as 'standard' real variables and strings. Another way of looking at the sort of statement that describes data is to regard it as not just a passive definition but as an instruction to 'organise' simpler data types. For example, the statement used previously, DIM B(20), could be read as an instruction to organise 20 variables into an array called B. Seen in this light such statements are often referred to as structuring the data.

At this stage it should be clear that the problem of finding an algorithm comes down to finding out what to do and what to do it to. However, once you have solved the task of what objects/data types you're going to use, the problem of what to do with them very often solves itself! Which is fortunate because there is very little specifically helpful guidance that can be given. There are, however, a number of general methods which will help you tackle a large problem and produce a program which is useful to other people. One of these methods, stepwise refinement, is dealt with below but, before we move on to it, there remains one last difficulty in programming that is worth discussing — background.

BACKGROUND KNOWLEDGE

It is surprising the way people expect programmers to move from one subject to another and still write useful programs. For example, professional programmers at various points in their careers might be asked to produce a stock control program and later find themselves working on a project involving graphical display in three colours. It is clear that, to make any progress with either problem, the programmer must first spend some time becoming familiar with the problem. The trouble is that all too often insufficient time is spent at this stage of building up a background in the area before starting to construct a program. The result is that the programs often work perfectly but do the wrong job or solve the wrong problem.

To sum up:

Difficulties in writing a program can come from three sources—
1) lack of knowledge and application of data structures.
2) lack of knowledge and application of problem solving methods.
3) lack of specific background information.

STEPWISE REFINEMENT

When you look at someone else's program or a program that you wrote so long ago you have forgotten all about it, the first thing to do is to try to get an overall feeling for what it is doing. You might identify the first twenty lines as an initialisation part, the middle as doing some calculation and the final part as output. Once you have this overall structure you can move on to seeing how each part does its job and find out how the calculation is done. Slowly your understanding grows as you identify the role of smaller and smaller parts of the program. Finally, you arrive at the point of view that the writer of the program must have had — you can see each instruction operating on every variable and know what each is for.

Now, let's return to the problem of writing programs rather than reading them. Instead of thinking of the program that you are trying to write as a long list of instructions, think of it as a collection of modules each doing part of the job of the whole program. This, of course, brings with it the difficulty of deciding how to split the program into modules but, once again, considering how someone understands a program usually suggests a method. When trying to formulate a program you start with an overview and work down to smaller and smaller modules. You could think about writing a program as trying to understand one that you haven't managed to write yet, so why not start at the top! For example, the problem of writing a chess program is so overwhelming that most programmers have difficulty starting. However, to start the ball rolling the first attempt at a chess program would be something like:

```
start game
play chess until end of game
give results of game
```

This may not seem like very much progress but we can now look at each module and try to 'refine' its definition. The next step, of course, is to attempt to reduce each of the smaller problems yet further. For example, our next attempt might be:

```
start game —
   print titles
   set difficulty level and who is white
   initialise board and other 'play' variables
play chess —
   get move
   record move
   analyse board
   make move
   end of game?
results —
   you win or I win message
```

This process can be continued until the program is complete. In fact, what normally happens is that the refinement continues until one of the modules can be written in BASIC and from then on, the refinement is carried out as part of program development on the computer.

This idea of taking a bigger problem and splitting it down into a number of smaller problems and then taking each one of the smaller problems and splitting them down further and so on is called ***stepwise refinement*** or ***top down programming.*** It has a great many advantages but the one which people find most helpful is that it gets you started ! In practice, the neat theory that programs are written by successive refinement is a little way from the truth. Even the most skilled programmer sometimes gets it wrong and has to backtrack. Sometimes a stage in the refinement throws light on earlier versions and a better method can be seen or, sadly, sometimes a stage in the refinement can demonstrate that the overall approach is incorrect and there is no choice but to go back and start again. Still, at least you will now know one way how ***not*** to do it! However, it is very rare that it is impossible to salvage some part of the program designed during stepwise refinement.

SUBROUTINES

While we were discussing stepwise refinement, the idea of a module was introduced as a way of grouping together instructions with a common purpose. It would be an obvious advantage if the computer language the program was being written in made some provision to keep this grouping and, if possible, made it stand out in some way. Various methods are possible but the only one available in standard BASIC is the ***subroutine.***

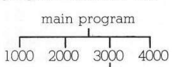
In BASIC you can collect together a list of statements and, as long as you end it with RETURN, you can treat it as a subroutine. The list of instructions can be referred to by writing GOSUB N where N is the line number of the first line in the list. Most BASIC programmers will recognise this description of a subroutine but might be a little confused by the way in which they are being likened to modules. Instead of writing the list of instructions out every time they are needed, a subroutine is constructed and GOSUB used instead. This is a valid reason for using subroutines but what we have discovered is that subroutines are useful even if the list of instructions is only going to be used ***once!*** This use of subroutines as modules makes a BASIC program which has been constructed by stepwise refinement reveal the stages it has been through. The first stage gives rise to a program which is often nothing more than a list of GOSUBs. The second stage of refinement produces the BASIC which makes up the subroutines used ▶

in the first stage and so on until all the subroutines have been defined. Thus, the stages of the stepwise refinement are frozen into the final structure of the program — the hierarchy of subroutines.

This sort of programming is often referred to as *top down modular programming* (TDMP for short!). The advantages of TDMP are immense — programs are easier to change, easier to understand and easier to debug. The only real disadvantage of this sort of program construction is that it doesn't give the most efficient version of the program. It is difficult to give an example to show clearly the advantages of TDMP because it only becomes apparent in medium to large programs — short programs are easy to write using any method! However, a short example might help to show what a TDMP program looks like. Consider the problem of reading in a string of characters and reversing its order.

```
  10  GOSUB 1000  Initialise
  20  GOSUB 2000  Get input string
  30  GOSUB 3000  Reverse string
  40  GOSUB 4000  Print result
  50  STOP

1000  B$=""       Clear output string
1010  RETURN

2000  PRINT "TYPE ANY MESSAGE"
2010  INPUT A$    Read in string
2020  RETURN

3000  FOR I=LEN(A$) TO 1 STEP-1
3010  GOSUB 5000  Get Ith character
3020  B$=B$+C$    into variable C$
3030  NEXT I      and add it to B$
3040  RETURN

4000  PRINT
4010  PRINT "OLD STRING=";A$
4020  PRINT "NEW STRING=";B$
4030  RETURN

5000  C$=MID$(A$,I,1)
5010  RETURN
```

The subroutine structure for this program looks like this:



You may feel that having subroutine 3000 call another subroutine (ie 5000) to extract the Ith character is going a little too far but, apart from illustrating the idea of the second stage of refinement, the use of subroutine 5000 makes conversion to BASICs that do not have the MID$ function a lot easier.

PARAMETERS

The short example in the previous section shows up a number of the problems using BASIC's GOSUB and RETURN instructions to write modules. Computer languages vary to the extent that they recognise the need to use modules — Pascal is very good and BASIC is terrible (indeed, BASIC's lack of a good method of forming modules is its biggest let down as a programming language).

The first problem with the BASIC GOSUB. . .RETURN is that it is impossible to give names to subroutines. Some versions of BASIC (eg BBC BASIC) do allow names to be given to a special form of subroutine (known as a procedure) but in most BASIC programs, the only real option is the extensive use of comments to make sure that the purpose of any subroutines used is obvious. One trick which can be used in any BASIC that allows expressions to be used in GOSUB statements is to define variables with the appropriate names and assign the correct line numbers to them. For example, in Sinclair BASIC (used on the ZX81 and ZX Spectrum) you can write things like GOSUB 2*I+56. So instead of GOSUB 1000, etc, you could write:

```
10  INITIALISE=1000
20  GETSTRING=2000
30  GOSUB INITIALISE
40  GOSUB GETSTRING
```

A second and more serious limitation on the BASIC subroutine is that there is no way of using parameters (a parameter is perhaps most familiar to BASIC programmers from its use in functions). For example, in the function definition:

```
DEF FNT(A,B)=A+B
```

the variables A and B are parameters. They have nothing to do with any variable of the same name in the rest of the program. The parameters simply show what the function is to do — ie add the first parameter to the second. When a function is used, real variables are substituted for the parameters. For example:

```
10  C=2
20  D=3
30  PRINT FNT(C,D),FNT(D,C)
```

The reason why parameters are so useful is two-fold — first the function can be written without worrying about what variables have been used in the rest of the program and second, the function can be used any number of times with different data. These advantages would be no less welcome as part of a subroutine facility.

Unfortunately, standard BASIC doesn't make any sort of provision for parameters in subroutines. It is true that one or two versions of BASIC (BBC BASIC again!) do provide extended subroutine facilities that include parameters but if you want to stay with standard BASIC, you have to either abandon the idea of parameters or settle on a system of naming variables within each subroutine.

Consider the string reversal example. Subroutine 5000 returns the Ith letter of the string A$ yet can be re-written in a form which does not depend on the particular variables I and A$.

```
5000  C50$=MID$(A50$,I50,1)
5010  RETURN
```

Using the subroutine is now a little more long winded:

```
3010  A50$=A$:I50=I:GOSUB 5000
3015  C$=C$50
```

but the subroutine can be used to extract the Ith character from any string variable and the answer can be stored in any other. The main advantage, however, is difficult to see in this 'static' example. Using the variables A50$ and I50, subroutine 5000 can be written ***before*** the rest of the program secure in the knowledge that no other subroutine will use its variables!

Mike James

ELEGANT PROGRAMMING – 2

Producing a working system using structured Lubalin programming.

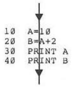
In the first article of this series we discovered that writing programs using modules was a good idea in that it provided a systematic way of constructing a program. What we have said nothing about so far is what sort of programming goes on *within* a module. If you are writing a large program then you can use the idea of a module, or subroutine, to break the problem down into a hierachy of smaller problems — but at some point you still have to write some statements that do something!

What this comes down to in practice is using IF statements to select between alternatives, FOR loops to repeat things and GOTO to get to other parts of the program. Notice that although it is possible to be precise about what IF and FOR are used for, the statement about the GOTO is very vague — why should we want to 'get to another part of the program'? The answer to this question will emerge in the course of this article but along the way we will meet a collection of ideas that are usually referred to as 'Structured Programming' or SP. So, if you've ever wondered what SP was all about — read on.

AN ITALIAN MEAL?

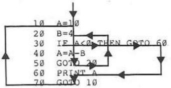
A program is executed one instruction at a time and at any moment the instruction that is being obeyed may be thought of as 'controlling' what the computer is doing. Thus, as a program runs, control is passed from statement to statement. This often referred to as the 'flow of control'.

In a BASIC program, the rule is that control flows through the program in order of increasing line number unless otherwise directed by a 'control' statement such as IF, FOR, GOTO or GOSUB. If you are given the listing of any BASIC program then you should find it possible to take a pencil and draw lines through a program to trace all the possible paths that the flow of control might take. In a very simple program with no control statements the flow of control line would go from top to bottom. Its 'shape' would therefore be a straight line:



```
10  A=10
20  B=A+2
30  PRINT A
40  PRINT B
```

As programs become more complicated and use control statements such as GOTOs, then the flow of control line becomes split and doubles back on itself and its 'shape' becomes complicated:



```
10  A=10
20  B=4
30  IF A<0 THEN GOTO 60
40  A=A-B
50  GOTO 20
60  PRINT A
70  GOTO 10
```

Neither of these two example programs does anything useful so don't think you are missing their meanings!

As you might expect, the more tangled and twisted the flow of control line becomes, the more difficult it is to understand the associated program. In BASIC the biggest cause of tangled flow of control is the GOTO statement. If you are used to writing programs 'as you go along' without planning then you will be both haphazard and opportunist in your use of GOTO. In other words you will use GOTO either to get you out of a situation that you have programed yourself into or you will suddenly see how you can save a few lines by jumping to another part of the program. If you go back and modify such a program then the chances are that you will make the flow of control even more tangled than it was in the first place!

The free use of GOTO in BASIC produces a complicated flow of control.

The result is that most BASIC programs resemble spaghetti. Unfortunately, this spaghetti image has tended to rub off on BASIC rather than on the programmers who produce it.

There is another important problem with spaghetti programs. If you look at any small section of a program then, if the flow of control is very mangled, it is almost certain that there will be more than one way of reaching it. Now this may not be a problem if the program is working, but if you're trying to debug it then it can cause unnecessary confusion. The value each variable in the section is supposed to have depends on two things — its initial value and what is done to it in the section. Although what is done to it is fixed and easy to see, if there is more than one way to reach the section of program then it is very difficult to work out its initial values because these depend on which route was taken.

If you agree that spaghetti programs are something that we could well do without then you will be interested to learn that there are some very simple rules that help produce clean, 'well-structured' programs. If you do not see what is wrong with spaghetti then much of what follows will at best seem irrelevant and at worst an unnecessary cramping of your programming style. The usual ▶

argument against SP is that 'free style' programming can produce the fastest and most compact programs. This ***is*** true. However, you ***can*** produce a program that is just as efficient by writing a well-structured program and then, by changing *only* the parts that are causing it to slow down, you can make it run at an acceptable speed.

STRUCTURED PROGRAMMING

The basic idea behind producing well-structured programs is very simple.

Use only a small number of ways of changing the flow of control.

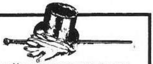
If we choose very simple ways of changing the flow of control and build up larger programs using these then the overall flow of control should be easier to follow for two reasons!

1) It should be easier to spot the standard 'shapes' that make up the flow of control line, and
2) There should be no 'tangles' in the flow of control line.

If, in addition, each of the ways of changing the flow of control can be put together so that each part of the program can only be reached by one route then so much the better.

At this point you may think that although this solution sounds simple it is impossible to use. Surely you cannot restrict the ways in which control can pass through a program to a few simple forms? After all, think how complicated the flow of control can be in a spaghetti program. Is it really possible to rewrite such a program so that it is well structured?

The answer to both of these questions and many similar ones is YES! And this is not just an opinion. It can be proved that you can write ***any*** program using a very small range of simple changes in the flow of control. In particular, ***any*** program can be written using only some form of IF statement, that will select between two alternatives, and some form of conditional loop, that will cause a group of statements to be repeated until a condition is satisfied. In terms of the flow of control, the IF statement divides it into two and the loop causes it to go back on itself.



You can write any program using a combination of conditional loops and IF statements.

Although you can write programs using only these two types of control it is easier in practice to invent a few more. Traditionally, four types of flow of control have been used to write structured programs:

1) The sequence (see Fig. 1) — This is the default flow of control in all computer languages and corresponds to instructions being executed one after the other.
2) The DO...WHILE (see Fig. 2) — This repeats a sequence as long as a condition, P, remains true. The test is made before the sequence block is entered. Therefore, it is possible to bypass the execution of the sequence at the first test.
3) The DO...UNTIL (see Fig. 3) — This repeats a sequence until a condition becomes true. Notice that in contrast to the DO...WHILE, the test is made after the sequence has been executed — hence the sequence must be carried out at least once.
4) The IF (see Fig. 4) — This tests some condition and carries out sequence 1 if it is true and sequence 2 it is false.

The choice of these four is sensible in that it includes two types of loop — one that can skip the sequence without ever allowing it to be carried out and one that insists that it is carried out at least once — and an IF

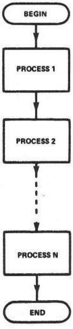


Fig. 1. A straight sequence.

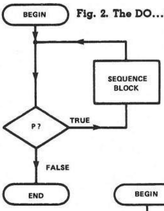


Fig. 2. The DO...WHILE loop structure.

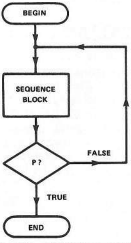


Fig. 3. The DO...UNTIL loop structure.

that will select between two alternatives. Using these statements you can build any program, no matter how complex.

Many computer languages contain this traditional set as direct instructions. For example, Pascal includes an IF, a WHILE and UNTIL statements. Other languages, for example many versions of BASIC, do not and as a result it is often said that it is impossible to write well-structured programs using them. This, however is not true. This traditional selection is just that — a *SELECTION.* As long as you include some sort of IF and some sort of conditional loop then you can choose your own set of control statements and use them to write well-structured programs.

In practice there are many advantages in using, or at least knowing about, the traditional set — they are the basis of many computer languages and are used by many other people.

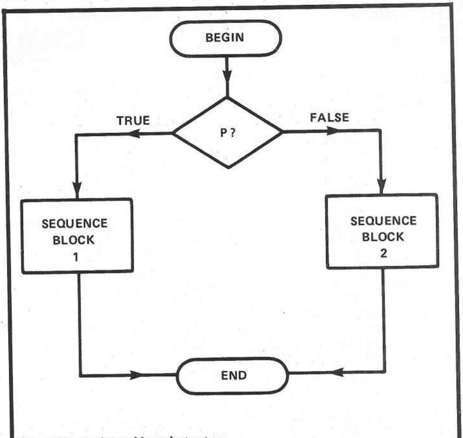


Fig. 4. The conditional branch structure.

STRUCTURING BASIC

The discussion about structured programming has so far been in general terms and seems to make no reference to any computer language in particular. However, lurking behind this discussion is the shadow of a computer language such as ALGOL or Pascal that includes the traditional structured control statements described above. What this means is that most descriptions of structured programming concludes with some statement like 'SP is not possible in BASIC'. Indeed, the argument that SP is a good thing is often incorrectly taken to mean that BASIC is a bad thing! But,

SP is a programming technique applicable in* any *computer language.

The easiest way of using SP in BASIC is to find equivalents of the traditional set of forms of the flow of control. This is not difficult. For example, a WHILE loop can be written as follows.

```
IF not condition THEN GOTO

other BASIC
statements

GOTO
end of loop
```

The UNTIL loop can be written as:

```
start of loop

BASIC statements
   . . .
   . . .
IF not condition THEN GOTO
end of loop
```

The standard form of the IF statement is more difficult because of the different ways that the BASIC IF is implemented. Using only the simplest form of the BASIC IF, ie IF . . .THEN GOTO, the structured IF of Fig. 4 can be implemented as:

```
IF condition THEN GOTO
BASIC statements
corresponding to
the ELSE part

GOTO
BASIC statements
corresponding to
the THEN part

end of IF
```

These days most BASICs come complete with a full

`IF ... THEN ... ELSE ...`

form of the IF statement which is the exact equivalent of Fig. 4. An important point to notice is that if the list of statements following the THEN or the ELSE is very long, it is advisable to turn them into subroutines.

Although you can write any BASIC program using just these forms of flow of control, it would be silly to ignore the very convenient FOR loop. You can use the FOR loop to repeat a list of statements a known number of times. SP does place a couple of restrictions on the way you can use a FOR loop, for example, you should not jump to a statement that is in a FOR loop nor jump out of a FOR loop before it is completed. ▶

THE 'ONE STATEMENT' IF

There is a simpler form of the IF statement which is well known to BASIC programmers. It is the one that carries out a statement if the condition is true and skips to the next statement if it is false, ie

IF condition THEN statement
rest of the BASIC program

This is just a special case of the general IF with no statement following the ELSE. However, this 'single statement' form of the IF is more difficult to handle than it looks. For example, the one statement IF can be constructed from the two statement form of the IF simply by not writing any statement in the ELSE part:

IF condition THEN GOTO
GOTO
BASIC statements
forming THEN part

end of IF

But this can be simplified by changing the condition to 'not condition' and leaving the THEN part empty:

IF not condition THEN GOTO
BASIC statements
forming the ELSE
part

end of IF

In BASIC it is easier to SKIP a list of statements when a condition is false than to carry out the list when a condition is true.

FOR EXAMPLE . . .

At this point, an example is long overdue. It might be thought that the best sort of example would be one that introduced two programs — one written in free style spaghetti and the other doing the same thing but well structured. In practice though, a program that is long enough to show the full horror of free style programming is too long for the newcomer to SP to appreciate what is going on in the structured version. Instead, a short program illustrating the use of the BASIC equivalents of the WHILE, UNTIL and IF will be given. If you want to see the transformation that SP can bring to a spaghetti program then I would urge you to convert an existing program and see for yourself how much more clearly you understand the program and appreciate the method.

Structured programming is an activity on-lookers rarely appreciate.

The example given below is a simple sorting program. Given an array A of size N, it sorts the contents into ascending order using a technique known as a bubble sort. The array is scanned and adjacent elements (ie A(I) and A(I+ 1)) are compared. If they are in the wrong order then their contents are swapped. The array is fully sorted when no swaps are made during a complete scan.

A structured BASIC program to implement this method is given below. Notice that the lines of BASIC that correspond to the traditional structured flow of control are indicated.

```
10  DIM A(N)
             :       :
             :       :
         50  K=0
         60  I=1
         70  IF I>=N THEN GOTO 170
         80  IF A(I)>A(I+1) THEN
             GOTO 110
U        90  I=I+1
N W     100  GOTO 160
T H I   110  T=A(I)
I I F   120  A(I)=A(I+1)
L L     130  A(I+1)=T
  E     140  K=1
        150  I=I+1
        160  GOTO 70
        170  IF K<>0 THEN GOTO 50
        180  ....REST OF PROGRAM
```

The IF in the middle of the program compares the two adjacent elements. If they are in the wrong order, the 'THEN' part swaps them and increments the index I. If they are in the correct order, then the 'ELSE' part simply increments the index I. The WHILE carries out the scan of the array A. As long as the index is less than N then the IF will be carried out (ie the loop continues WHILE I<N). The UNTIL is responsible for the repetition of the scan until no swaps are made. The variable K is set to zero inside the UNTIL and remains zero unless a swap occurs when it is set to 1. Thus, if a swap is made, K is not zero and the scan is carried out again.

There are a number of things that can be done to improve the readability of this program. For example, as we know how long the array A is, the WHILE loop can be replaced by a FOR loop. That is, instead of line 70, write:

```
70  FOR I=1 TO N-1 .
```

and remove lines 90 and 150 and replace 160 with NEXT I. Remember that this program is an illustration of SP methods and not necessarily the only way it should be written.

PROGRAMMING STYLE

Now that you know something of structured programming you should try to write some programs taking particular care over how you handle the flow of control. As you gain experience, you do not have to feel that you are forced to apply structured programming as if it were a straight jacket. What is important is that you understand the ideals of good programming style. Within these ideals you are free to develop your own methods. Do not value tricks that take a week to explain to another programmer, but try to make your programs clear and easy to understand. If your programs are easy to understand there is much less chance that a logical error will remain undetected for long. Remember, bugs love spaghetti!

ELEGANT PROGRAMMING – 3

It's bug-hunt time as our series takes a look at crash-proofing programs.

So far in the series, we have considered what makes good and elegant programs from a programmer's point of view. In other words, we have been looking at programming as an activity but we have not considered the product in much detail. If you use a logical and ordered approach to programming then the program you produce will be better than if you hadn't. Elegant programming methods do not guarantee a program that a user will admire, only a program that the programmer will admire!

What makes an 'elegant' program from a ***user's*** point of view? The term 'elegant' applied to a program means a wide variety of things to whoever is using it. Clearly how fast and how small a program is are aspects of program efficiency. Just about everything else is a matter of elegance. Two programs may carry out the same job using roughly the same amout of time and memory but deliver entirely different degrees of service to the user. One may 'crash' (ie not complete the job) given only a small amount of user ignorance or perversity, whereas the other may allow the user the liberty of completely ignoring any instructions concerning the 'proper use' of the program.

Other important aspects of elegance are not so obvious. Take, for example, maintenance and extendability. A program may do the job in hand today but what about tommorrow's job? Some programs can be modified easily, others are such a nightmare that it's preferable to start from scratch! These and other aspects of elegance are mainly about using a good programming method and testing the product well, but this is not true of the degree to which a program is crash proof.

A good program is well written and well behaved!

In the early days of computing, the demand was for a program to solve a specific problem and if the result was a program that worked if it was treated with care then everything was fine. Users had to be content with what they got! These days there is no excuse for producing a scrappy program along with a list of do's and don'ts. Indeed the average user demands a program that not only works in the sense of getting the right answer but a program that does not crash no matter what is thrown at it.

One reason for this is that the average user is becoming less technical as computers find their way into domestic applications. This means that programs have to be good due to the intolerance of the non-programmer. We can all imagine a typical scenario — the computer game fanatic kicking his machine to pieces because just before he managed the score of 1,000,000,000 (something only achieved once in a lifetime), the message:

```
***** INTEGER OVERFLOW IN 17839 *****
** SYSTEM ABORT FUNCTION INITIATED **
```

appeared on the graphics screen.

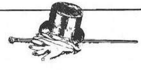
Another reason is that the number and quality of programmers is increasing. This has led to an awareness that things do not have to be quite so bad.

ELEGANCE VS EFFICIENCY

Some programmers would argue that elegant programs are all very well but an efficient inelegant program is much better than an elegant program that takes a week to get its results. This is a valid but misleading argument. It is too often used in order not to bother about elegance. The claim that a better job would have been done if more machine power were available is really no more than an excuse. This is thought to apply to micros in particular because everyone knows that they are not very powerful. However, the following points should be kept in mind:

1) A single user micro can be more powerful than a heavily loaded time share system.
2) The bulk of most conversational programs are not speed critical.
3) The time consuming parts of any program can normally be isolated and treated separately.

As computer hardware becomes cheaper and more powerful, the importance of efficient programming becomes less and perhaps in the limit, vanishes. In other words:



Efficiency is a hardware problem — programming is about elegance.

Remember that you can double the speed of any program simply by using a double speed microprocessor but the same factor of two is normally ***very*** difficult to get by programming alone.

The reason the above maxim is only close to the truth and not entirely true is due to the ▶

existence of an area of theoretical computer science/logic known as complexity theory. You can show that for some problems, if you use the wrong algorithm the problem can take as long as the universe has left to it to work out and most algorithms are the wrong algorithm. In this sense, programming is also about finding a good algorithm.

CORRECTNESS, BUGS AND CRASHES

Assuming a program is using a method that will in theory do the job required, there are a number of reasons why it might fail:

1) There may be syntax errors — this is equivalent to a spelling mistake.
2) The method might not be implemented properly, ie the program may not be correct. This is equivalent to not saying what you intended to say.
3) The program might fail because of some user generated condition — this is equivalent to being ambiguous.

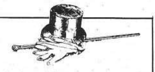
The first two are what most programmers will recognise as 'debugging' a problem. The first problem is easy to deal with because your friendly interpreter or compiler will let you know what is wrong. The second program is very difficult to give any general advice about as it all depends on what alogorithm you ***should*** be using versus what algorithm you ***are*** using! You can also find that you are using the wrong algorithm for a wide variety of reasons ranging from just not knowing the correct algorithm to using the wrong variable at some point in a program because of a typing error. The third reason is a subtler but more common reason for a 'fully debugged' program crashing in use. It is worth examining why.

When FORTRAN was introduced as the first high level programming language, it started an important trend — programming by default. For example, the letters I to N were taken by FORTRAN to mean integers unless otherwise instructed. A language such as ALGOL or Pascal would demand that the programmer declare that I,J,K, . . .N were integers (or something) before they were used. Thus, in ALGOL-like languages nothing is assumed about variable names, but in FORTRAN-like languages (including BASIC) variable names are assumed to be one type unless otherwise instructed. The effect of this is that in one case programmers have to build something into their algorithm to deal with variable types and in the other, it can be ignored.

Programming by default seems harmless enough and indeed as long as it's confined to this sort of thing, it is positively an advantage; it saves a lot of time when writing a BASIC program to ignore declaring all the variables before they are used. The trouble with programming by default is that, without realising it, it has managed to work itself into the way an interpreter/compiler implements your algorithm!

ERRORS BY DEFAULT

The most dangerous area of programming by default, because most programmers rely on it, is the automatic definition and handling of run time errors. Look at the error message lists of your BASIC interpreter — each one indicates an error condition that in a good program should never happen. If one of them does occur then the sequence of action which results is usually not under your control. To summarise:



The detection of run time errors and the subsequent actions are usually part of the definition of the interpreter or compiler and are forced on your program.

To make all this clear, let's consider the innocent looking problem of inputting data. Normally, all that is involved is writing INPUT in a box in a flow diagram and then translating it to an INPUT statement in BASIC. However, INPUT is a very complex operation and there are a large number of ways of interpreting it. For example, what is to be done if the input is of the wrong type? Most interpreters will give a cryptic error message and ask for the input again, some (the worst) simply give the error message and stop! The implementation of INPUT on the PET is particularly frustrating for a beginner — if you type a Return without any inputs the program stops! This is clearly programming by default!

In the first generation of BASIC interpreters, error handling was very poor. Most people felt lucky that they had a BASIC interpreter at all and so handling errors was a small problem. As time passed, things got better and more error detection was built in. The trouble was that the writers of interpreters think that most errors are best handled by them!

If we examine how errors come about it's possible to define two types:

1) Inner errors — these are intrinsically difficult to detect within the language being used. For example, it is difficult to discover if arithmetic overflow is going to ***occur*** before the execution of an expression.
2) Outer errors — these are detectable, in theory, at some point in the program before they occur. For example, in theory, it is possible to check that a file exists on disc before attempting to read it and generating a FILE NOT FOUND type of error.

In an ideal world, the writer of an interpreter would transform all inner errors to outer errors and allow the programmer to try to detect them. Of course, if they are not detected the interpreter must still attempt to deal with them without crashing the program, ie programming by default is not ***all*** bad.

CRASHING IS UNNECESSARY

Now that we have recognised the problem in all its complexity, the solution is obvious. Rather than allow the interpreter to program parts of our algorithm that we haven't bothered about, we must take control! This is easy to say but often very difficult to do. In some cases, it turns out to be impossible because of the poor quality of the BASIC interpreter. There are three broad methods of programming at the level of detail required, however, and some combination of the three should solve most problems.

1) Smaller steps. In the example quoted, programming by default came about because of the different ways in which the INPUT box of the algorithm could be expanded. If, instead of using the blanket term INPUT, we had taken the trouble to define our requirements in detail then (if we are lucky) no crashes should occur. Thus, our first solution is to program ambiguous parts of an algorithm in smaller, more precise steps. This depends on there being a way of breaking the action down into smaller steps and not all dialects of BASIC will allow this. For example, in PET BASIC (and many others), the statement GET will return a single character typed at the keyboard without waiting for a Return or checking it in any way. (If no key has been typed then a null string "" is returned.) This is obviously the smallest input action that can be defined in common BASIC. Using this simpler instruction, it is possible to build uncrashable input routines. For example, the following short program will read in positive numbers.

```
 10  GET A$
 20  IF A$="" THEN GOTO 10
 30  IF A$=CHR$(13) THEN GOTO 100
 40  IF A$<="9" THEN GOTO 70
 50  PRINT "DIGITS ONLY PLEASE!"
 60  GOTO 10
 70  IF A$<"0" THEN GOTO 50
 80  I=I*10+VAL(A$)
 90  GOTO 10
100  PRINT I
```

Lines 10 and 20 get a character from the keyboard. Line 30 checks to see if Return has been pressed and if it has transferred control to line 100 to print the result. Lines 40 to 70 check that the character typed is a digit and line 50 prints a message if it isn't. If the character is a digit then it is converted to numeric form using the VAL function and added to the running total entered so far in I. Although this program is virtually uncrashable (see later for how to crash it!), it is far from friendly. For one thing it doesn't allow you to edit any number that you enter. Friendly uncrashable input routines can become very long and have to be very comprehensive. (See MAXIMANDER in ***Personal Software***, Vol 2 No 1, Summer 1983 for an example.)

2) Use prechecks. It is usually easy to detect an outer error before it happens. The problem is what to do if you succeed in detecting an error. For example, suppose you check the existence of a disc file before trying to read it — what if it's not there? If the program should have created it earlier, the only reasonable conclusion is a machine error — but what if someone has changed the disc during the run or started the program somewhere other than at the beginning. A possible answer is to ask the user if it's a good idea to start again. One thing is certain, however — it is not a good idea to print FILE NOT FOUND and stop the program! If the user was supposed to supply the file name then a polite message suggesting that the user 'have another go at getting it together' is often not enough. I have often been stuck in the middle of a program with the prompt NO SUCH FILE — INPUT FILE NAME AGAIN appearing each time I type another guess at the file name! It may be that the user cannot remember the file names so it is a good idea to offer the chance of looking at the disc's catalogue.

Taken to extremes, acting on the detection of an outer error can border on an exercise in artificial intelligence. So remember, if life gets tough, ask the user what to do — he might after all have some non-artificial intelligence!

ON AN ERROR?

It is possible to pre-check for most inner errors but in general it isn't easy. For example, the input routine given earlier is only almost uncrashable — if you try to enter a number that is larger than the machine can hold it, ***will*** crash with a cryptic message from the interpreter. You might think that you can avoid this problem by adding the lines:

```
 75  IF MAXNUM-(I*10+VAL(A$))<0 THEN
     GOTO 110
 :   :                :                :
 :   :                :                :
110  PRINT "NUMBER TOO BIG"
120  I=0
130  GOTO 10
```

to the program where MAXNUM is the largest number that the computer can handle. The trouble with this solution is that the program will still crash when the interpreter works out the expression in the IF statement. It doesn't matter even if the whole expression works out to less than the largest number—if you reach the largest number in the course of the calculation then the program crashes! The correct solution is to change line 75 to:

```
75  IF (MAXNUM-VAL(A$))/10<I THEN
    GOTO 110
```

This is the same test but it cannot cause a crash because the expression on the left of the '<' sign is certain to be smaller than MAXNUM.

3) ON ERROR GOTO. Some dialects of BASIC convert a lot of inner errors to outer errors by using an ON ERROR GOTO statement. In BASIC, the ON ERROR GOTO statement is unique in that it doesn't do anything when it is encountered during the running of a program. Following an ON ERROR GOTO 'line number' statement, any error detected by the interpreter causes a GOTO or a GOSUB to that line number. Hence, every error is detectable by the user and is an outer error. (Unfortunately, some BASICs define a set of

▶

errors that always cause interpreter controlled handling — this is unnecessary.) Most micro BASICs do have an ON ERROR statement; the notable exception being PET. The way that these work varies quite widely but most supply an additional variable called ERR or ERROR which contain the code of the error that has occurred and a variable called ERL or ERRLINE which contains the line number of the line that the error occurred in. Using these two variables it is possible to decide what to do about the error. After the error handling has been completed, control can be passed back to the main program by the RESUME statement which will continue execution at the line the error occured or by RESUME 'line number' which will continue execution at the specified line number.

The trouble with using an ON ERROR statement is similar to that of dealing with detected outer errors. A very simple program becomes quite complex if full error handling is included. For example, the very simple addition program:

```
10  INPUT A
20  INPUT B
30  PRINT A+B
40  GOTO 10
```

becomes very long if you use ON ERROR to crash proof it.

```
  5  ON ERROR GOTO 100
 10  INPUT A
 20  INPUT B
 30  PRINT A+B
 40  GOTO 10
100  IF ERR=6 THEN GOTO 200
110  IF ERR=13 THEN GOTO 250
120  IF ERR=23 THEN GOTO 200
130  PRINT "AN ERROR HAS OCCURRED
     WHICH SUGGESTS A HARDWARE
     FAULT"
140  STOP
200  PRINT "THE NUMBERS ARE TOO BIG
     TO CARRY OUT THE ARITHMETIC"
210  IF ERL=30 THEN RESUME 10
220  RESUME
250  PRINT "PLEASE TYPE A NUMBER -
     NOT LETTERS"
260  RESUME
```

If any error occurs while the program is running then control is transferred to line 100. The variable ERR is checked to see what error has occured and various messages are generated to inform the user. (To explain the error codes — 6 is overflow, 13 is type mismatch and 23 is linebuffer overflow.) Notice that the action taken depends on the error code and the line number that the error happened in. Also, the only condition that causes the program to stop is when the error is totally inexplicable and could only be due to a machine fault.

If a BASIC has an ON ERROR statement then there is no excuse for any interpreter generated error messages.

In the next article of this series we move away from programming techniques to look at some very necessary background information — randomness and its use in programs.

THE VALLEY
© ASP LTD 1982

What are you... Barbarian or Wizard?

VERSION AVAILABLE! CBM 64

Choose your character type carefully... Barbarians recover quickly but their magic doesn't come easily. A Wizard? Slow on the draw and slow to mature...but live long enough and grow wise enough and your lightning bolts are almost unstoppable...

The Valley is a real-time game of adventure and survival. You may choose one of five character types to be your personal 'extension of self' to battle and pit your wits against a number of monsters. Find treasure, fight a Thunder-Lizard in the arid deserts of the Valley, conquer a Kraken in the lakes surrounding the dread Temples of Y'Nagioth or cauterise a Wraith in the Black Tower. In fact, live out the fantasies you've only dared dream about. BUT BEWARE... more die than live to tell the tale.

You've read the program (Computing Today — April '82 ... Now buy the tape. Tape versions (£11.45 each inc P&P and VAT) available for: ZX Spectrum (48K), Atari 400 and 800 (32K) Dragon, BBC Model A and B, Sharp MZ-80A, VIC-20 (with 16K RAM pack). Disc version (£13.95 each inc P&P and VAT) available for: Apple II (DOS 3.3), Sharp MZ-80A and PET 8032 (8050 drives). Full instructions are included with the game, but if you want more detail on the program, a 16 page reprint of the original 'Computing Today' article is available at £1.95 all inclusive.

Fill in the coupon and return it to:

ASP Software,
ASP Ltd,
145 Charing Cross Road,
London WC2H 0EE

and become one of the many to play... The Valley...

Please send me the following versions of
The Valley Tape.........@£11.45 all inclusive of P&P and VAT.
Disc.........@£13.95 all inclusive of P&P and VAT.

I enclose my cheque/Postal Order/International Money Order (delete as necessary) for:
£.....(Made payable to ASP Ltd)
or Debit my Access/Barclaycard
(delete as necessary)

Please use BLOCK CAPITALS
NAME(Mr/Mrs/Miss)...
ADDRESS...
.... POSTCODE.....................
Signature... Date.....................
Please allow 21 days for delivery

TRADE ENQUIRIES WELCOME

ELEGANT PROGRAMMING – 4

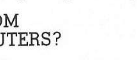
Our series advances into the world of uncertainty and randomness. Can you compute the odds on a successful return?

After spending so much effort in previous parts of this series describing how to go about writing programs that behave predictably, it might seem counter-productive to spend this episode considering how to make programs ***unpredictable!*** However, since the start of the personal computer boom, playing games has been a major occupation and most computer games involve an element of randomness.

Randomness in programs may be something that a user takes for granted but it can pose quite a problem even for the most experienced programmer. The trouble is that using randomness in programs depends upon knowing about something other than programming — ie probability.

RANDOM COMPUTERS?

The idea that a computer can be random in the same way as a thrown coin or a rolled dice is a strange one. A computer is a very complex mass of electron circuitry but at no point is its operation 'vague' in the same way as the behaviour of a thrown coin. If you know a computer's state at any moment, you can predict its state at any time in the future. This very fact is the basis of program debugging — for if we couldn't predict what the computer ***should*** do, how would we ever know that a program was behaving 'not as expected'! The first law of programming is that:



A working computer is absolutely predictable

and its obvious corollary is that 'an unpredictable computer is a broken computer'!

All this talk of absolute predicability seems to widen the gulf between computers and randomness rather than reconcile them. However, the key to the problem lies in the previous suggestion that the flight of a coin was 'vague'. The toss of a coin is entirely predictable — in theory at least. In practice, its behaviour is so complicated that it is beyond the powers of most people to predict how it will land.

This sort of randomness is different from most people's idea of randomness. A truly random event is one that is unpredictable in theory as well as in practice. An event that we regard as being random just because it is too difficult to predict deserves a different name. 'Pseudo-randomness' is the term that has been coined to cover this class of happening (pseudo meaning false or imitation.) Most of the things that we think of as being random are in fact pseudo-random although sometimes it's difficult to decide. The point is that, in practice, there is little difference between randomness and pseudo-randomness. If something is too difficult to predict, it matters little that it is theoretically possible to predict! You may be wondering at this point if there is anything that is truly random and theoretically impossible to predict. The answer is yes, but you have to look into the realm of atomic physics before you find it!

PSEUDO-RANDOM NUMBERS

The idea that you might bet on a single number can be extended to the case where a computer produces a whole list of numbers which are unpredictable (rather like tossing a coin more than once). If this list or sequence of numbers is going to prove useful as a source of randomness then it must be, for all practical purposes, unpredictable. Now the trouble is that most of the sequences of numbers which come out of a computer are predictable or at the very least they show general patterns of behaviour, ie they tend to increase and decrease in regular ways.

A sequence of numbers is said to be psuedo-random if:

1) there is no 'practical' way of predicting the next number in the sequence, and
2) there are no 'patterns' of any sort in the sequence that could be used to 'guess' the next number in the sequence.

This definition sounds reasonable enough — the only trouble is that it is entirely at the mercy of the words 'practical' and 'patterns'. What one person might find practical may be another's impossibility! And as for the word 'patterns' well . . .!!!

PRANGING IT

The emphasis has now moved away from producing random events or happenings to something a little more abstract — a pseudo-random sequence of numbers. As we shall see later, given a pseudo-random number sequence you can make any event happen seemingly at random with any given probability, so producing such a sequence is of great practical importance. A program which produces pseudo-random numbers is often called (very reasonably) a Pseudo Random Number Generator or PRNG for short. It is very much more difficult than you might think to write a PRNG. The condition about the number not being ▶

easy to predict is simple enough — for example, how many people can compute SIN(x) in their heads?! The other conditions cause the problem because most calculations, no matter how complex, tend to produce sequences of numbers which show a regular pattern.

One of the first PRNGs, the mid square method, was suggested by the mathematican and computer scientist Von Neumann in 1951. It is fairly easy to use but it has a tendency to produce numbers of the form 00XY and XY00 periodically. Each number in the series is determined by squaring the previous number in the series and throwing away all but a fixed number of the middle digits of the result. For example, if we are generating four digit random numbers and the last number was 5069, the next number is given by squaring 5069, giving 25694761 and then take the middle four digits ie 6947. The following short BASIC program implements the mid square method:

```
10  INPUT "STARTING VALUE";A
20  A$=STR$(A*A)
30  A=VAL(MID$(A$,INT(LEN(A$)/2),4)
40  PRINT A
50  GOTO 20
```

(STR$ is a function which converts numbers to strings and MID$(string, I, J) extracts the substring of length J starting at the Ith character.)

CONGRUENTIAL PRNGS

Although the mid square method serves as an illustration of how a complex calculation can produce a pseudo random sequence, it's not the best or the most popular method used today. This position is held by the so-called 'congruential' method. Although this method has the sort of name that might make you hide behind your computer, it is not much more difficult to understand than the mid square method. A congruential generator produces the next number in the sequence by multiplying the previous result by a constant and then finding the remainder after dividing by a second constant. The quality of the random numbers produced depends very much on the choice of the two constants used and it varies from hopeless to excellent! The BASIC program given below implements a congruential PRNG which was used on one of the first computers — ENIAC.

```
10  INPUT "STARTING VALUE";A
20  A=A*23
30  A=A-INT(A/100000001)*100000001
40  PRINT A
50  GOTO 10
```

In this case, the first constant is 2 and the second is 100000001. Line 30 works out the remainder when A is divided by the second constant.

One other feature of congruential PRNGs is that the quality of the random numbers that they produce depends on the starting value as well as the two constants. For example, to see a **very** non-random sequence try entering 0 in answer to line 10 in the above program!

RND AND RANDOMISE

If you know BASIC at all well you might be wondering what all the fuss about generating random numbers is about. Nearly every version of BASIC includes a function called 'RND' which can be used to produce random numbers in the range from zero to one and is as easy to use as SIN or COS. This is, of course, very useful but it does tend to lull one into a false sense of security — of course the numbers are random, they were produced by RND weren't they ?!? The sad fact of the matter is that:

RND doesn't always produce numbers that are random enough

The BASIC random number generator isn't any different from the PRNGs that we have been considering — they are just as fallible. The big advantage of the RND function is that, as part of BASIC rather than a program written in BASIC, it is very fast.

On a more practical level, RND presents a problem to any programmer trying to write programs which can be run on different versions of BASIC. The trouble is that there is no standard for the meaning of any parameters which are allowed in RND. Some versions of BASIC don't allow any parameters (eg Sinclair BASIC), some need a −1 as a parameter to produce random numbers (eg Microsoft) and others allow the parameter to control the range of numbers produced (eg BBC BASIC). The safest thing to do is to assume that RND will produce numbers in the range 0 up to but not including 1 (I don't know any full versions of BASIC for which this isn't possible) and just face up to the fact that when converting programs, you may have to change all your RNDs to RND(−1)s or vice versa.

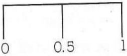
The BASIC function RND should seem perfectly straightforward by this point, but most BASICs also have an associated command 'RANDOMISE' which seems to be an unnecessary complication. A lot of BASIC manuals recommend that if you want ***really*** random numbers you should always use the RANDOMISE command before using the RND function. This often strikes programmers as a rather strange idea — either the numbers from RND are random or they are not and if they are, how can RANDOMISE make them more random? The answer to this sort of question should now be obvious by thinking about RND as just another PRNG. If you look at either of the two BASIC programs given earlier, you will see that you are invited to give a value to start the sequence off.

This value is often referred to as the 'seed' because the rest of the random sequence 'grows' from it. If you start a PRNG off using the same seed then you will get the same sequence of numbers. The BASIC function RND is no different from any other PRNG and it needs a seed to start it off. Just think for a moment where this seed might come from. Ideally, the seed should be random because we don't want to use exactly the same sequence of numbers every time the computer is switched on. This would make games boring to say the least! However, where do you get a random number **from** to start the random number generator off?

The answer is that the RANDOMISE command starts the random number generator off with a seed that is obtained from an area of the machine's memory where the value is constantly changing. What area of memory this is, varies from machine to machine — in most it contains a number related to how long the machine has been switched on. (Sinclair users note that everyone else's RANDOMISE is your RANDOMISE 0.)

USING RND

Given that the BASIC function RND is good enough (and for most applications, especially games, it is) how do you go about using it to produce random events? Let's suppose that we need to choose between one of two things that a program might do such that each is equally likely. RND returns a number in the range 0 to (but not including) 1 and as this sequence is pseudo-random, each number in this range is equally likely to 'come up'. If you consider the interval from 0 to 1:



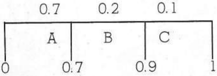
you should be able to see that the number that RND produces is equally likely to fall into the range from 0 to 0.5 as it is from 0.5 to 1. In other words, RND 0.5 will be true about half of the time and this is exactly what we need. So the statement:

```
IF RND<0.5 THEN <action 1> ELSE
<action 2>
```

will cause the program to 'do' action 1 and action 2 at random and roughly equally often. This leads to the simplest and perhaps most often written random program:

```
10  RANDOMISE
20  IF RND<0.5 THEN GOTO 50
30  PRINT "HEADS"
40  GOTO 60
50  PRINT "TAILS"
60  PRINT "PRESS ENTER/RETURN FOR
    ANOTHER THROW"
70  INPUT A$
80  GOTO 20
```

This is fairly straightforward but the problem which causes most difficulty is in extending this 'two-event' case to making a number of things happen possibly with different rates or probabilities. In fact, this is not at all difficult once you realise that the probability of getting a value of RND in any interval is proportional to the length of the interval. For example, suppose we want to a program to choose between one of three different alternatives A, B and C, such that A should happen 10% of the time, B should happen 20% of the time and C the remaining 70% of the time. If the interval between zero and one is divided up in the following way:



then the number produced by RND will fall in the interval A, 70% of the time and in B, 20% and in C, 10% of the time. This is exactly what we need and translating this into the most economical set of IF statements gives:

```
10  R=RND
20  IF R<0.7 THEN GOTO <action a>
30  IF R<0.9 THEN GOTO <action b>
40  <action c>
```

The logic behind this program is that the first IF detects R smaller than 0.7 and the second IF detects a value of R smaller than 0.9 (but it only gets the opportunity to do so if the first IF was false, ie if R was greater than 0.7). This means that the second statement following the second IF is only carried out if R is greater than 0.7 and less than 0.9 which is exactly what we want.

In general, if you have N different events and the first must happen P1 of the time and the second P2 and so on to PN, then you can program this as:

```
R=RND
IF R<P1 THEN <action 1>
IF R<P1+P2 THEN <action 2>
IF R<P1+P2+P3 THEN <action 3>
.......
```

RANDOM RANGES

Sometimes, rather than using RND to choose between a number of possibilities, it is useful to change the number produced by RND to lie in a range other than 0 to 1. For example, if you want to write a program which will mimic the roll of a dice you could use the method give in the last section with six equally likely events or you could change the 0,1 range of RND to 1 to 6. This is just a matter of simple arithmetic. If you want to generate a random number in the range from a up to but not including b then use:

```
RND*(b-a)+a
```

You can test that this works by working it out with values of zero and just less than one for RND. If you want to produce integers (whole numbers) from a to (and including) b, use the following:

```
INT(RND*(b-a+1)+a)
```

For example, in the case of the dice program, a is 1 and b is 6 so:

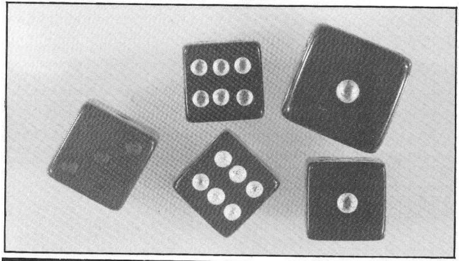
```
10  PRINT INT(RND*6)+1
20  GOTO 10
```

will print random integers in the range 1 to 6.

TESTING RANDOMNESS

Any program which involves randomness is very difficult to fully test. The reason for this is two-fold. First, if any of the parts of the program are carried out only very occasionally, you may wait a long time before you see them in operation. Second, even if you do see a section of the program in operation and an error comes to light, it may be difficult to repeat the exact conditions that caused it. There are no complete solutions to these problems and as a consequence, you have to allow ▶

extra time for debugging any program which contains randomness. There are two things that you can do to help with the problem, however. You can use RANDOMISE or a particular parameter value in RND (which depends on the version of BASIC that you are using) to set the value of the seed at the start of the program. This will force the random number generator to always start from the same value and means that you can repeat any errors by re-running the program. Another technique which is worth trying is to replace the RND function by constants that will force the program to go through each and every section in turn. In practice, there are often good reasons why testing 'modified-easy-test' versions of a program is not possible, eg it destroys any interactive aspect of a game, and so testing by extensive playing is the only option!



A SERIOUS SIDE

Although the emphasis has been on using randomness in games, the time has come to point out that there are other uses for the RND function. Most computer games are based upon some aspect of the real world. A program which plays a card game, for example, is copying something that happens with real cards using nothing but programming logic. This idea of modelling, or simulating, can be extended to things other than games. For example, you could write a program to simulate the way a nuclear reactor works and find out the best way to run it by 'playing' with the program. If you think that this is far-fetched then all I can say is that putting the real world inside a computer is one of the growth areas of computing.

By now you should have realised that we can structure programs! But, what on earth are data structures? To solve this elementary riddle you'll need to read the next article.

IF YOU HAVE A BBC MICRO
THEN YOU NEED
Laserbug

Laserbug is the newsletter of the Independent National BBC Microcomputer Users Group. If you want the best source of information on the BBC Micro you can't do without Laserbug No matter what your interest – hardware, software, business, games or education then Laserbug has something for you.

Also, Laserbug has available many special offers including dust covers (for computer, monitor, printer, disks), cassette leads and 1.2 ROMS FOR ONLY £5.50 INCLUSIVE – THE CHEAPEST PRICE ANYWHERE! (Members Only)

Laserbug defies description – send off for a sample copy and you'll find that it sells itself to you. See one and you'll be hooked for life!!!

Please supply me with
[ ] more details about Laserbug and your special offers
[ ] a sample copy for £1.00 and an A4 SAE (17p postage)
[ ] 1 UK 12 Month Subscription for £12.00
[ ] 1 UK 6 Month Subscription for £6.00
[ ] 1 Overseas Surface Mail Subscription for £14.00
(air mail rates on application)

Please send the goods to:
NAME .... ADDRESS ....
....
I enclose a cheque/PO for £ · p made payable to LASERBUG.
Please send the form to LASERBUG Dept. C, 10 Dawley Ride, Colnbrook, Slough, Berks., SL3 0QH.

SOFTWARE FOR YOUR CBM 64

Scramble	**£7.00**	**Matrix**	**£8.00**
Frogger	**£7.00**	**Star Trek**	**£7.00**
Sprite Man	**£7.00**	**Rox**	**£4.95**
Crazy Kong	**£7.00**	**Grid Runner**	**£8.00**
Panic	**£7.00**	**Pakacuda**	**£5.75**
Lander	**£9.95**	**Cyclons**	**£5.75**
Attack of the Mutant Camels	**£8.00**	**Escape M.C.P.**	**£5.75**
		Centropods	**£5.75**

A NEW ADVENTURE FOR YOUR 64
Dead Man's Gold £9.00. *We dare you to seek the treasure and return it to the correct grave.*

NEW ADVENTURE FOR YOUR VIC 20+16K
The Enchanted Chalice £8.50. *Find the chalice if you can.*

NEW FOR ANY SPECTRUM
Race Fun, Quackers, Escape M.C.P., Centropods, Frogger, all at only £5.75, and for 48K Spectrum: Phantasia £5.75, Hidden City £5.95

Now in stock from "Imagine Software" for any Spectrum —
JUMPING JACK £5.50
This is one game we just can't stop playing.

Cheques/P.O.s to:

BYTEWELL,
203 COURT ROAD, BARRY, S. GLAM. CF6 7EW.
Tel: (0446) 742491

Mike James

ELEGANT PROGRAMMING –5

Writing structured programs requires the use of the correct data structures.

The idea that a program consists of two parts — actions and objects — was introduced right at the beginning of this series. However, up to this point we have only examined the way the actions could be specified. In other words, we have only looked at the subject of program structure. Now it is time to consider the other side of the coin — data structures. Although data structures have been left until after a discussion of program structure, this shouldn't be taken to imply that they are any less important. Programs are easier to construct if the programmer has as wide a range of alternative data structures as possible at his or her fingertips.

There are two distinct ways of producing 'objects' for programs to operate on. First, there is a range of simple or fundamental 'data types' which programs can use for calculations, etc. Second, there are ways of taking these fundamental data types and putting them together to make ordered arrangements. Ways of arranging simple data types into larger objects are usually called 'structuring methods'. For example, a character is a fundamental data type, but an array is a structuring method because you can define arrays of numbers or characters. In this article, the subject of fundamental data types will be examined paving the way for a discussion in the next article of data structuring methods.

REPRESENTATIONS

As most programmers know but often forget, the only sort of data which a computer can deal with directly takes the form of a binary number. The reason why it is so easy to forget this elementary fact is that it is the purpose of a high-level language to extend the range to include more exciting and useful forms of data and so hide the strict limits of the hardware from its programmers. This is a good thing because the largest binary number which most micros can deal with in one go is only eight bits or, in other words, in the range 0 to 255. Clearly, to be able to do anything useful, it is necessary to use this very limited form of data to represent every type of 'object' that we would like to write programs about. Although this sounds like an awesome task there are in fact, only two fundamental types of data and everything else can be produced by organising these. The two fundamental types correspond to the two types of numbers in common use — integer and real numbers.

Integers are the simplest type of number that we use on a regular basis. An integer is a whole number; for example, 3, -3 and 0 are all integers wheras 3.14, -3.8 and 3.0 are ***not*** integers. This all seems easy enough apart from 3 being an integer and 3.0 not being an integer because, surely, 3 and 3.0 are equal and therefore the same? The point is that two quantities being equal doesn't imply that they are the same in every way. A real number is one that has a fractional part — and 3.0 is a real number with a fractional part of zero! If you find this confusing or hair-splitting then perhaps it is better to think of a real number as one which has the possibility of having a fractional part.

The idea of integers and real numbers is something that most programmers meet in one form or another even in high-level languages. However, it is quite possible to use some versions of BASIC and never be troubled by the difference between numbers with fractional parts and numbers without fractional parts. Indeed, one of the strongest of BASIC's many good points is that if you don't want to know about such things then you can ignore them almost completely. The way this is achieved is by making every variable capable of storing a real number and then treating integers as real numbers which just happen to have zero fractional parts!

It's beginning to sound as if introducing two different types of number into computing is unnecessary and that other languages should follow BASIC. This is to some degree good sense, in that there is a lot to be said for protecting the user from unnecessary complications. However, even BASIC has to admit that it is sometimes important to be able to convert a real number into an integer and normally provides the function INT which will round a real value down to give an integer. Also, there are many occasions where fractional values are logically unacceptable; for example, it makes no sense to write TAB 3.4 where a TAB moves the current printing position, ie the cursor, to a specified column — obviously column 3.4 doesn't exist.

FUNDAMENTALLY . . .

Integers are the most fundamental of all the data types and are the starting point for everything else, so it's not surprising that they cannot be avoided completely. For

▶

example, if you want to extend the range of data types to include the usual printed characters, then the only way that this can be done is to assign an integer to represent each different character. Such assignment of integers to characters is known as a character code (probably the best known of which is the ASCII code) but it is important to realise that it is not the only one in use.

No matter what you may be led to believe, you cannot store a character inside a computer — merely an integer that represents it.

The pair of BASIC functions CHR$ and ASC (or CODE in some dialects) make the connection beween integers and characters clear. CHR$(I) returns the character whose code is the integer I, and ASC(C$) returns the code of the character stored in C$. Apart from just being a way of storing characters inside a computer, the way that integers are assigned to characters also governs the results of comparisons. For example, "A"< "B" is true *only if* ASC("A")< ASC("B") is true. In other words, the order of the character set is simply a reflection of the order of the integers in the character code.

SCALARS

In the same way that integers can be used to extend the range of data types to characters, they can also be used to represent other equally simple 'objects' — the scalars. A data type is a scalar if the values that it can take on are 'countable'. You may at this point find the idea that a data type can take on values which are un-countable a little difficult to comprehend but, as we shall see later, this is entirely possible.

The only two scalars which most versions of BASIC recognise are characters and the Boolean data type which has the two values 'true' and 'false'. A Boolean data type is the result of any sort of comparison. For example, 'X>0' is either true or false depending on what is stored in X. As with the characters, the values true and false are stored by assigning each one to an integer. Which integers are used depends on the dialect of BASIC, but 0 for true and − 1 for false is quite common. Some versions of BASIC try to cover up this use of integers to represent Boolean data by not permitting you to use the results of Boolean expressions in arithmetic or PRINT statements but many will allow you to write things like:

```
PRINT (2=3),(2=2)
```

which will first print the integer which represents false and then prints the integer which represents true. (It is worth trying this line of BASIC as a simple experiment to see what your version of BASIC makes of it.) An unwanted side effect of the use of integers to represent 'true' and 'false' is that, in the same way as the characters 'take up' the order of the integers which represent them, so do the two values true and false. Let us suppose that true is represented by − 1 and false by 0 (the representation used by Microsoft BASIC -80). In this case, 'false' is greater than 'true' and if 'x' and 'y' are a pair of comparisons which are either true or false then we can draw up the following table to show the results of x not equal to y.

x	y	x<>y
false	false	false
false	true	true
true	false	true
true	true	false

If you look closely at this table you should be able to recognise the truth table for exclusive OR!

WEEK ENDING

BASIC may not provide other scalars as standard data types but BASIC programmers certainly make use of them. For example, if you want to write a program that records the day of the week that something happens to you, you might start by assigning a number to each day, ie Monday = 1, Tuesday = 2 and so on. By assigning integers in this way you are creating your own data type —$ days of the week. Some computer languages, Pascal for instance, have special facilities allowing the introduction of new scaler data types in such a way that you can write satements such as:

```
day=friday
```

It is sometimes said that one problem with BASIC is that it cannot cope with user-defined scalars in this way and has to resort to statements such as:

```
day=5
```

instead. In fact, it is simple write programs in BASIC which make user-defined scalers easier to understand by defining variables with appropriate names and values. For example, (assuming the version of BASIC you are using can handle long variable names) given the following list of definitions:

```
10  MONDAY=1
20  TUESDAY=2
..  .....
..  .....
70  SUNDAY=7
```

you can write something like:

```
100  DAY=FRIDAY
```

and:

```
110  IF DAY=MONDAY THEN PRINT "First
     day of the week"
```

This simple use of variables to store the integer codes assigned to each value of the new scalar type gives BASIC nearly everything which languages such as Pascal have except the automatic checking for nonsense such as:

```
day=8
```

The above statement is something which an application program should pick up before it happens anyway! Using this method you can even write things like:

```
10  FOR I=MONDAY TO FRIDAY
20  PRINT "WORKING DAY"
30  NEXT I
```

or even

```
10  FOR DAY=MONDAY TO FRIDAY
20  PRINT "WORKING DAY"
30  NEXT DAY
```

The data type scalar — encompassing integers, characters, Boolean and any user-defined scalars — is the most commonly encountered and most useful data type in BASIC or any programming language. Indeed, it is difficult to think of applications which don't use scalars apart from those involving nothing but long numerical calculations.

BACK TO NUMBERS

The importance of scalar types is a reflection of the fact that, contrary to popular opinion, computers don't spend much of their time doing 'different sums' but are more often busy moving non-numeric data from one place to another and making decisions. However, it would be a mistake to ignore the problems involved in using numbers and doing arithmetic on computers completely. Even in a high-level language it is helpful to know what is going on!

A more difficult problem is the storage of numbers ***smaller*** than zero — in other words, negative numbers. There are many ways of extending the range of integers to include negative numbers, but all of them involve a little mathematics to be fully understood. Fortunately, all the BASIC programmer has to be aware of is that to accommodate negative numbers, it is necessary to split the range of numbers which can be stored into two — the first half is considered positive and the second half negative.

Storing and using integers is only complicated by needing to use negative numbers and needing a large enough range to avoid overflow errors. However, storing real numbers is considerably more difficult. The point is that if you can store an integer then you can store it exactly, but no matter what scheme you choose to store real numbers there will always exist numbers that you cannot cope with.

For any two real numbers there is ***always*** *another which lies between them.*

This is clearly not true for integers — after all what integer lies between 2 and 3! This observation should also indicate that real numbers are not simple scalars as it is not possible to count how many reals there are in the same way that you can count how many days of the week there are. Another difference is that for a simple scalar there is the concept of 'next', ie the next day of the week, the next integer after 2 (ie 3), and so on; but for reals, there is no such concept of 'next'. Consider for a moment, what is the next real number after 2, is it 2.1, or 2.01. or . . .? All of this should convince you that there are integers and their associated family of simple scalar types and there are the reals, and both present their own particular set of problems.

REPRESENTING REALS

Before going on to consider the sort of problems that real numbers cause in everyday BASIC (and other high-level languages) it is worth spending a little time considering how real numbers can be represented inside a computer. The most obvious way of representing a real number is to change an integer to a real by assuming that there is a decimal point written after a particular bit of the number. For example, if you assume that an eight-bit number is in fact made up of four bits, a decimal point and followed by another four bits, then 10101011 would represent a real number given by 1010.1011. Notice that the decimal point isn't stored inside the computer, we just remember where it is when interpreting the contents of a memory location. Although the dot written in the middle of the number has been referred to as a decimal point, it is more correctly called a binary point!

If you try to work out what real number 1010.1011 represents, you should have no trouble with the first part as 1010 is easily converted to 10 using the usual method for changing from binary to decimal, but what about the part following the binary point? In the same way that the values increase by a factor of two for every place to the left of the binary point, they decrease by a factor of two for every place to the right of the binary point:

```
8 4 2 1   ½ ¼ ⅛ 1/16
1 0 1 0 . 1 0 1 1
```

This gives a value of $8+2+\frac{1}{2}+\frac{1}{8}+\frac{1}{16}$ for the entire number, or in other words, 10 and $\frac{11}{16}$ or 10.6875. This sort of representation is known as 'FIXED point' and it was very common in the early days of computing. However, it suffers from the problem of not being flexible enough for general use. The trouble is that it simply cannot cope with the range of numbers use in calculations, especially scientific calculations.

The solution is to abandon the usual decimal point notation altogether in favour of the so-called 'exponential' or 'scientific' notation. This separates a number into two parts, the first — the exponent — giving the overall magnitude of the number, and the second — the mantissa — giving the most significant digit of the number. In normal use a number is written in exponential form as:

```
'mantissa' E 'exponent'
```

and can be converted to the more usual decimal form by:

```
number=mantissa*10^exponent
```

Where ∧ is to be read as 'raised to the power of'. So .43E3 is .43 ⋆ 1000 or 430 and .321E-2 is .321 ⋆ 0.01 or .00321.

Using this format a wide range of numbers can be represented. When used as a method of storing real numbers inside a computer it is usually called 'floating point' representation rather than exponential, but it essentially the same. The exponent is stored as an integer in one memory location and the mantissa is stored as a fixed point number in several other locations. The exact details of how this is done varies from BASIC to BASIC but all that should concern the BASIC programmer is:

— how many digits or bits are used for the mantissa

and:

— how many digits or bits are used for the exponent.

These two factors govern the accuracy and range of numbers which a BASIC program can handle. For example, Microsoft BASIC-80 uses three memory locations for the mantissa and one for the exponent — this gives about seven digits of precision and a range of about 10^{38} to 10^{-38} (both of which I find difficult to imagine!).

PUT TO GOOD USE

The fact that some BASIC's don't even bother to distinguish between integers and reals has already been mentioned, but this doesn't mean that we can throw caution to the wind and treat reals and integers in the same way. Some versions of BASIC do provide a range of different types of number and hence variables. For example, both Microsoft and BBC BASICs provide integer variables (indicated by a '%' sign at the end of the variable's name). As any arithmetic with integers is a lot simpler than for reals, the use of integer variables is often to be preferred to reals. To see if your BASIC works faster with integer variables, time the following programs:

```
10  K=0
20  K=K+1
30  I=K*K+K/K
40  IF K<1000 THEN GOTO 10
```

and:

```
10  K%=0
20  K%=K%+1
30  I%=K%*K%+K%/K%
40  IF K%<1000 THEN GOTO 10
```

There are other advantages to using integer variables apart from speed. In particular, if you are trying to write programs which handle money, it is comforting to know that integer variables will keep track of the last penny!

More seriously, it is important to realise that real variables cannot carry out calculations exactly. For example, try the following program:

```
10  K=0
20  K=K+1/7
30  IF K=1 THEN STOP
40  GOTO 20
```

On most versions of BASIC this program will never stop although you would expect it to end after adding 1/7 to K exactly seven times. To see why, add the line:

```
25  PRINT K
```

and run the program again.

Because of difficulties with accuracy, you should never compare real numbers in IF statements. Instead of:

```
IF A=B THEN ....
```

use:

```
IF ABS(A-B)<C THEN ....
```

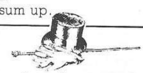
Where C is small enough to ensure that A and B are close together when the condition is true and yet large enough to ensure that the condition is true when the difference between A and B is about the same as the accuracy they are stored in.

IT ALL ADDS UP

A second, less publicised problem with floating point numbers reveals itself when you try to take the difference between two large numbers that are roughly the same magnitude or add together two numbers of very different magnitudes. In the case of the subtraction, the result you get will have more to do with the error involved in representing two large numbers.

To see this for yourself try 1E20+ 1E10 — in many case you'll discover the answer given is 1E20. This is rarely a practical problem, but a great many quantities of interest between two large numbers. For example, many BASIC programs which calculate the standard deviation of a list of numbers give results which are almost random when the average of the numbers is large and the true standard deviations is small.

To sum up,



Real numbers deserve to be treated with more caution than they usually are.

APPLE II USER'S GUIDE
TRS-80 ASSEMBLY LANGUAGE MADE SIMPLE
Program
Pocket Guide

ELEGANT PROGRAMMING – 6

Following from the previous discussion on data types we take a look at the methods used to structure them.

Last time we looked briefly at some of the fundamental data types that can be used to construct programs. Even if we added all the obscure data types to this list we would still lack such familiar things as arrays and strings. To produce the wide variety of data 'objects' that are essential to programming we have to discover ways of organising the fundamental types into 'data structures'. In other words we need to define ways of 'structuring data'. The best way of understanding this idea is by looking afresh at a well known example.

THE ARRAY

Because of the way that programming languages are taught it is quite common for programmers to miss the idea of constructing new data items by applying a structuring method ot existing data types. For example, the idea of a one-dimension BASIC array is often introduced as a completely new idea. It is described in terms of the rules for using it and what it can be used for. However, a one-dimensional array is nothing more than a 'collection' of real variables with a special way of naming each variable. The traditional way of thinking of this collection is as if the variables were lined up in a row (or column!). The whole collection is given a name – the array – and in order to refer to any of the variables that make up the array it is necessary to supply a two-part name. The first part is the name of the array, there may be more than one array in a program, and the second part specifies which variable within the array you are referring to. This second part of the name is often called the 'qualifier' and the whole two-part name is 'qualified name'. This idea of a qualified name will crop up time and again when learning about structuring methods.

This rather academic description becomes a lot more familiar when related directly to the one-dimensional array that we have been considering. The array name is (in most versions of BASIC) a simple variable name – A, for example. The qualifier comes from the idea of the variables lined up in a row. Any variable can be picked out by giving its place in the line. The only problem is what to call the first variable – 0 or 1. Some versions of BASIC call the first variable 0 and others 1 but, for the sake of simplicity, we will assume that the first variable is called 1. Thus the qualifier part of the name takes the form of a number starting at 1 and ending with the number of variables in the array. A qualifier that takes the form of a number is more often called an 'index' so the usual way of writing the name of a variable in an array is:

array name (index)

For example, A(5) refers to the fifth variable in the array called A.

Once you have realised that forming an array is a structuring method it is possible to use the same method to produce a wider range of arrays than are normally present in BASIC. The first thing that can be done is to allow other types of fundamental variables to be organised into arrays. For example, you can line up integer variables to form integer arrays and character variables to form character arrays (not to be confused with strings). No matter what fundamental variable is used to form an array the basic idea is the same – to reference a single element you have to use a qualified name.

A less obvious way of extending arrays is to allow the type of the qualifier to be something other than a number.

For example, A(wed) would refer to the variable that was used to store something to do with Wednesday. In general it is possible to imagine arrays where the qualifier part of the name is any scalar (see last time's article in the series) apart form a real number. In practice BASIC ignores this extension and restricts qualifiers to integers only. This is no great disadvantage because if the method for constructing scalar types suggested in Elegant Programming – 5 is used, expressions such as A(mon) = 56 will work because 'mon' is a simple variable holding an integer.

ARRAY EXTENSION

The most important extension to arrays arises from the observation that a structuring method isn't restricted to ordering fundamental variables but can be used to order data types produced by an earlier application of a structuring method. For example, applying the structuring method that results in a one-dimensional array to organise a collection of one-dimensional arrays results in what is more often called a two-dimensional array. Each one of the data items that is lined up in a row is itself a one-dimensional array consisting of a line of real variables. Now, to refer to one of the real variables it is necessary to supply two qualifiers (indices). The first indicates which element of the first array is required and as this element is itself a one-dimensional array, a second qualifier is needed to indicate which real variable in this array is needed. If a two-dimensional array A consists of a line of three data types, each of which is itself an array of five real ▶

variables, then A (2) (4) is the fourth element in the second constituent array. Some versions of BASIC allow this doubly quaified name to be written as above (ZX BASIC, for example) but most demand that the indices are written within one set of brackets and separated by a comma; ie instead of A (2) (4) you must write A(2,4). This difference in notation hides the fact that:

A two-dimensional array is not something new and different, it is simply the result of applying the same structuring method twice

The traditional way of thinking about two-dimensional arrays as a table of variables with each individual identified by a particular row and column number also hides this fact. However, if you notice that the table can be thought of as a row of columns (or ***vice versa***) you should be able to see the connection between one-dimensional arrays and two-dimensional arrays.

IMPLEMENTING ARRAYS

The job of actually creating an array is normally carried out by whatever BASIC interpreter or compiler you happen to be using. All you have to do is write DIM A(10) and an array of ten elements is created for you to use. However, it is sometimes useful to be able to create arrays directly. For example, if you want to use an array with 1000 elements where each element will only be used to store a number in the range 0-255 then you can save a lot of storage by implementing your own array. If you simply write DIM A(1000) then BASIC will allocate 1000 variables for you to use and each variable will be capable of holding numbers of a much greater range than you need. A real variable will often use four memory locations to store a single number — thus A(1000) typically uses around 4K of memory. However, if your version of BASIC gives you some way of reserving memory for such things as machine code routines etc then you can implement the array A(1000) directly into 1K — a quarter of the usual storage. The method is simply to use POKE to store values in the elements of the array and PEEK to recall them. The only problem is finding which variable corresponds to which memory location. If we suppose that the start of the memory area is stored in the variable START and the index of the required element is held in I then the address of the memory location is simply START+I−1. (The '−1' is necessary because the first element has an index of 1 and its address is START.) To store VALUE in A(I) we use:

```
POKE(START+I-1),VALUE
```

To recall the value in A(I) we use:

```
VALUE=PEEK(START+I-1)
```

The expression START+I−1 gives the address of the memory location used to hold the value corresponding to A(I). Because this expression relates elements of the array to a particular memory location it is referred to as a Storage Mapping Function or SMF for short.

The idea of an SMF can be extended to include cases where each element takes more than one memory location to store. If each element takes N memory locations to store and the first one is numbered 1 the SMF is:

```
START+(I-1)*N
```

which, if N=5, gives START+5 for the address of the second element in the array.

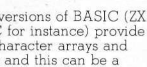
Using what we have already learned about one- and two-dimensional arrays it is easy to extend this to an SMF for a two-dimensional array. If the array is to be DIM A(X,Y) and each of its elements takes only one memory location to store then the SMF is

```
A(I,J)=START+(I-1)*J+J-1
```

If you look carefully at this SMF you should be able to see that the first part is the address of an element of a one-dimensional array where each element is J memory locations long.

STRINGS AND THINGS

As the above discussion of the way that the BASIC array can be extended (using any data type to form the elements and any scalar type to form the indices) was followed by the admission that these things cannot be done in BASIC, you might imagine that BASIC is lacking in data structures. While this is true it might come as something of a surprise to discover that the humble BASIC string is a very advanced (and necesary) data type that isn't present in some very sophisticated languages — standard Pascal, for instance. If you've followed the discussion of arrays in any detail you may be wondering what the fuss is about — surely strings are just one-dimensional arrays of characters. Well, it is true that one-dimensional character arrays are often used to perform the same job as strings but they are not strings.



The key difference is that an array of characters has a fixed length but a string can change its size.

Some versions of BASIC (ZX BASIC for instance) provide both character arrays and strings and this can be a constant source of trouble to beginners. For example, in ZX BASIC DIM A$(10) results in a character array being formed with exactly 10 elements. This means that you can write things like A$(3)="Z" which stores Z in the third element of the array. But, if you print now out the array using PRINT A$ you will get all 10 character variables printed even if you have only ever stored one letter in the array. However, the string B$ behaves in an entirely different way. Just like the character array, it is a collection of characters but it alters its size as the number of characters stored in its changes. That is, B$="Z" doesn't just store Z in the string variable B$, it also creates the storage space necessary to hold *one* character. If you had written B$="XYZ" then enough storage would have been allocated to hold *three* characters. You can, therefore, think of a string as a character array that has a variable number of elements. A data type that can change its size while a program is running in the way that a string can is known as a 'dynamic variable'. In general dynamic variables are more difficult to implement than static variables and so many programming languages ignore them.

The similarity between character arrays and strings is something that is not often brought out by BASICs. One exception is ZX BASIC where individual elements within strings and character arrays can be accessed by specifying an index, ie B$(3) is the third character in B$ whether B$ is a string or a character array. Other versions of BASIC, Microsoft for example, don't even allow character arrays and treat strings as fundamental data types. To access the Ith element (character) in a string you have to use a function such as MID$(A$,I,1).

As before, the structuring method that produces an array can be applied to character arrays or strings to produce two-dimensional character arrays or arrays of strings. The differences between these superficially similar data structures once again stems from the variable length property of strings.

RECORDS

Now that the idea of an array has been explored in detail the idea of a data structuring method should be easy to understand. Unfortunately, the array is the only real data structuring method that BASIC possesses. However, it is worth looking around to see what other languages have to offer and BASIC lacks. In particular it is worth knowing something about the 'record', a structure related to the array. Even though BASIC does not contain any way to construct records it can make programming easier to ***think*** in terms of records.

Before computers records were kept on paper and contained entries such as a person's name, address and telephone number. In other words, a record is a collection of variables of different types — the name and address are string variables but the telephone number is an integer. Some languages, Pascal for instance, provide facilities to define such collections of different variables under a single name. One way of thinking about this is to imagine a record as a one-dimensional array where each element can be a different data type. For example, the name/address/telephone number record might be defined something like:

```
RECORD  person = name : string
                 address : string
                 telenumber : integer
```

(The details of such a definition obviously depend on the language being used but the general form remains the same.) The name of the whole record is 'person' and this plays much the same role as the array name. Each element of the record is known as a field — thus 'person' is a record with three fields — 'name', 'address' and 'telenumber'. Each field has an associated data type that defines what sort of information can be stored in it — thus 'name' is a field that can store a string of characters while 'telenum' is a field that can only store integers. The only problem left is how to specify which one of the fields is being referred to. The answer is once again to provide a two-part name consisting of the record name and a qualifier. In this case the qualifier is the name of the field separated from the record name by a '.'. So, we can write things like:

```
person.name="mike"
person.telenumber=123
```

In the same way that the array structuring principle can be applied twice to yield two-dimensional arrays a record can contain a field that is another record! For example, the record 'person' defined above could have a field called 'bday' that is itself a record —

```
RECORD  bday = day : string
               month : string
               year : integer
```

To refer to an element in bday it is necessary to use a doubly qualified name, as in the case of a two-dimensional array. For example

```
person.bday.year
```

gives the year tht someone was born.

If this sounds interesting, now is the time to recall that most versions of BASIC don't support records! However, you can still use collections of strings and numeric variables with similar names as if they were records.

DYNAMIC DATA STRUCTURES

There is one class of data structures that can be added to BASIC by any programmer and, surprisingly enough, these are all dynamic data structures! The most important of these are the 'stack' and the 'queue' but there are others such as 'linked lists' and 'trees'. The trouble with describing these data structures is that if you've never wanted to use one it is difficult to see how they could be useful. However, if you don't know that they exist then there are some programs that, unless you re-invent the wheel, are very difficult to write.

The best known, and simplest, of all the dynamic data types is the stack. A stack is an ▶

area of storage that can be accessed through just two operations PUSH and PULL. PUSH stores an item onthe stack and PULL retrieves an item from the stack. The more formal name for a stack, 'Last In First Out' or LIFO, gives a clue to the order in which a sequence of items PUSHed on the stack will be recalled by PULL operations. The best way of visualise a stack is to imagine each item PUSHed onto the stack as being placed on the top of all the earlier items PUSHed onto the stack with a PULL operation always removing the topmost item. The last item to be PUSHed onto the stack is the first one to be PULLed off.

A stack is easy to implement in BASIC using a one-dimensional array and a single variable. The array is used to hold the items on the stack and the variable is used as a pointer to the current top of the stack. For example, to reverse three numbers is easy:

```
 10  DIM S(10)
 20  LET P=1
 30  FOR I=1 TO 3
 40  INPUT N
 50  LET S(I)=N
 60  LET N=N+1
 70  NEXT I
 80  FOR I=1 TO 3
 90  LET P=P-1
100  LET N=S(P)
110  PRINT N
120  NEXT I
```

Line 10 sets up the array to be used as the stack. Line 20 sets the variable P to 'point to' the top of the stack, in this case the top of the stack corresponds to the first free location in the array. Lines 50 and 60 form the PUSH operation. The number is stored in S(P) and then the pointer is incremented to point to the next free location. Lines 90 and 100 form the PULL operation and this should be easy to understand as it is similar to the PUSH operation.

That's all there is to implementing stacks and apart from worrying about PULLing data off the stack when there isn't any more and PUSHing more items on the stack than the size of the array allows there is nothing else to do. The most common application of stacks is in language processing, ie compilers etc. However, this is probably due to the fact that programmers who write compilers tend to know about stacks. In general a stack is useful whenever the order in which things have to be processed is different form the order in which they arrive.

QUEUING FOR IT

Once you have understood the idea behind a stack then a queue is just one step further on. The data structure called a queue mimics the behaviour of people queuing. A queue has a first person and a last person. People join the queue at the rear and leave the queue from the front. In the data structure called a queue the addition of data items happens at the rear and the retrieval of data items happens from the front.

The easiest way to implement a queue in BASIC is to use two pointers in place of the stack's one. The first pointer indicates the front of the queue and the second pointer indicates the end of the queue. In addition to these two pointers we can associate two new operations with every queue — JOIN and LEAVE. If F is the pointer to the front of the queue and R is the pointer to the rear then the two operations in BASIC are as follows:

JOIN

```
Q(R)=DATA
    R=R+1
```

LEAVE

```
DATA=Q(F)
    F=F+1
```

Thus the front of the queue moves relentlessly up the array, the trouble is — so does the rear! In order to stop the array having to be enormous you have to employ the extra trick of making the queue circular. If either of the pointers goes past the top of the array they are rest to point to the beginning of the array. The two operations now become:

JOIN

```
Q(R)=DATA
    R=R+1
    IF R>TOP THEN R=1
```

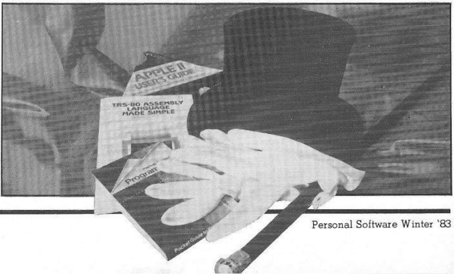
LEAVE

```
DATA=Q(F)
    F=F+1
    IF R>TOP THEN F=1
```

The queue starts off empty and the two pointers 'point' to the same place. This can be used to detect when the queue is empty. If after removing an item the pointers point to the same element of the array then the queue is empty. However, if this happens after an item has been ***added*** it means that the queue is full. The best way to find out how queues work is to dry run a queue on paper.

A program is part control structure, part data structure. If you select an inappropriate data structure then the program's control structure will be more complicated than it need be.

Data structures are important and in this brief introduction we have examined some of the elementary ideas involved. There is no doubt that the best way to learn about data structures is to use them creatively in actual programs — so experiment!



Mike James

ELEGANT PROGRAMMING – 7

Our series on advanced programming techniques takes a sideways step this time into the world of graphics. It's really moving stuff!

There is no way that a programmer can avoid using graphics on today's crop of microcomputers. Indeed, many a micro is dedicated to the production of graphics displays of all kinds. The trouble is that most programmers have to learn the necessary techniques by trial and error. If you turn to text books for information about graphics programming, you'll find that most of them concentrate on the more 'serious' aspects of graphics such as three dimensional representations etc. What you are unlikely to find information about is the comparatively crude static graphics used to 'brighten' an otherwise dull screen presentation or about the special techniques needed to produce synchronised dynamic graphics of the sort used by video games. This lack of information is probably due to the difficulty in producing any sort of theory of graphics programming.

In practice a good graphics programmer achieves his end product by a mixture of ad hoc rules and methods and a sense of good design.

It is possible to learn something about the *ad hoc* rules and methods but if you don't have a sense of design the best thing to do is to ask someone who has what they think of your product and take notice of their comments.

GRAPHICS HARDWARE

Before going on to consider the software side of graphics it is necessary to consider the various ways that computers handle graphics. Rather than give a complete list of every possible technique for producing graphics, which although interesting would take a lot of space, it seems more useful to limit the description to the few methods used by microcomputers. The world of microcomputer graphics is divided according to whether the electronics for the graphics facility is 'memory mapped' or 'port controlled'.

Memory mapped graphics is the most common way of providing graphics on micros because it is cheap and easy to use. The key feature of memory mapped graphics is that a portion of the computer's main memory is assigned to the production of a screen display. The contents of this area of memory is 'converted' by the computer's video circuitry to an image on the screen in such a way that a single memory location controls what is displayed at a particular area of the screen. The exact way that this happens varies quite widely but it is possible to distingush two general approaches — block character graphics and pixel graphics. If each memory location controls an area of the screen so that what it displays depends on a code stored in memory location then this is block character graphics. If however the pattern that appears on the screen depends on the pattern of 0's and 1's stored in the memory location then this is pixel graphics. (Pixel being short for picture element.)

Port controlled graphics are far less common in the microcomputer world. Instead of allocating an area of memory in the main machine to graphics an alternative approach is to build a separate 'graphics device' with its own memory etc and allow the computer to communicate with it over an interface port of some description. A typical example of this approach is a graphics VDU which can be attached to any computer via a serial port. Port controlled devices have two main advantages — they do not use any of the computer's main memory and, in principle at least, they can be connected to any computer. Their principle disadvantages are cost and speed. Such a device duplicates many facilities already in the computer and communicating over a port is usually slow compared to memory mapped

Memory mapped	—	each memory location controls what is displayed on an area of the screen
Port controlled	—	What is displayed on the screen is controlled by sending codes and other information over an interface port
Pixel graphics	—	a correspondence between the pattern of 0's and 1's stored in a memory location and what appears on the screen
Block graphics	—	No correspondence between the pattern of 0's and 1's and what appears on the screen. The content of each memory location is treated as a code that determines what will be displayed.

Table 1. First select your control mechanism and then the character type. Most common micros use memory mapped screens nowadays but more and more are turning to pixel graphics in the search for higher resolution displays.

▶

access. Because of the high cost of port controlled graphics you would think that you would never come across such a thing with a micro. In practice however what tends to happen is that a machine will have a video section built into the same box that has its own memory etc and communicates with the 'main computer' through a fast port or a small area of common memory.

All this classification of graphics systems is a little difficult to absorb at first so perhaps Table 1 might help. Real micros often use schemes that are not wholly described by any of the above categories. However it is normally possible to say that a micro's graphics is more like one than the other.

A SOFT VIEW

The above discussion is only of interest to programmers in so far as the hardware affects the graphics software provided. Because port controlled devices vary so much it is really only possible to describe the software techniques normally used with memory mapped graphics.

The easiest sort of graphics for a beginner to use are block graphics. The reason for this is that block graphics are handled in exactly the same way that alphanumeric characters are. If you know how to print the letter 'A' at a particular position on the screen then you can also print a special graphics character at the same place on the screen.

From a programmer's point of view block graphics look like an extended character set that can be PRINTED on the screen

Thus no new graphics commands are required to produce block graphics displays. All you need is the ability to select the correct graphics characters and position them on the screen to make up the 'shape' that you require. For example you may draw a box on the screen by finding the correct set of lines and corners among the character set and printing them in the right place. Drawing with graphics characters is rather like the old pastime of using a mechanical typewriter to make pictures!

Finding the correct characters to make up outlines of objects etc can be quite difficult until you are familiar with the character set provided. Indeed if a particular character that you require is missing then you will find that you cannot draw everything you want to. Offset against this lack of complete freedom to draw anywhere on the screen is the existence of solid graphics characters such as the four card symbols (heart, diamond, spade and club). The range of such graphics characters varies according to the particular machine you are using but it can be very wide and include such oddities as tank and spaceship shapes, guns, people, animals etc. The ability to print an entire space ship in one go at a specified screen location is very useful if you are writing games programs but not so appealing if you are trying to plot a graph!

Pixel graphics on the other hand are ideal for drawing graphs and other complicated outlines. As each point on the screen is associated with a particular bit in memory it is possible to alter any area of the screen independently of the rest. This gives rise to commands such as:

```
PLOT x,y
```

which (in many versions of BASIC) will make a point appear on the screen at a position specified by x and y. Using this command it is possible to draw any shape you require — straight line, circles etc. However, to make life even easier many versions of BASIC include some extra commands to draw a line or circle directly. The main problem with pixel graphics is that if you want to plot a solid shape such as a space ship or one of the card suit symbols it can take a large number of commands. At worst you will need one PLOT type command for every point in the shape and this is often as many as 64 or even 100!

Pixel graphics are good for drawing large outlines. Block graphics are good for small solid shapes.

As with all things in computers the subject of block and pixel graphics is not clear cut. For example the ZX81 uses block graphics but has a PLOT command. The BBC Micro and the ZX Spectrum both use pixel graphics but, because they have software provided for user-defined graphics they can be treated like block graphics machines. Using software it is possible to have the best of both worlds — except for one case . . . Some machines use the block character method to generate screens full of text but then use memory mapped pixel method for graphics. On such machines it is difficult to mix text and graphics without a great deal of extra software effort. (For example, in text mode the Apple uses block characters but in hi-res graphics mode uses pixel graphics).

THE SCREEN MAP

Although the memory mapped machines provide commands such as PLOT to enable the manipulation of individual bits in memory and hence individual points on the screen it is often the case that a more direct approach works better. It is possible to alter the contents of ***any*** memory location using the BASIC command:

```
POKE address,data
```

which stores the value 'data' in the memory location at 'address'. This means that it is possible to alter the contents of memory in the screen area directly and bypass the supplied graphics commands.

If you know that the screen occupies memory from 'START' to 'FINISH' then you might like to try the following two programs:

```
10  FOR I=0 TO 255
20  POKE START,I
30  NEXT I
```

and:

```
10  FOR I=START TO FINISH
20  POKE I,46
30  NEXT I
```

The first of the programs stores all the possible bit patterns into the first memory location of the screen area. This will give you two pieces of information — what place on the screen the first memory location corresponds to and what each bit pattern produces. Very often the first memory location corresponds to one of the four corners of the screen but this is not always the case. If your machine uses block graphics then each bit pattern will produce a whole new character on the screen. If your machine uses pixels, then you will only see eight points at most change. The second program POKEs a marker character into each memory location. If you watch the order that they appear you can determine how the memory locations correspond to the screen positions. For many machines it is possible to give a very simple formula for the way the memory location corresponds to screen positions and this is known as a screen map. It is important to notice that this is not always possible because the correspondence can be very complicated.

If you know the screen map for your machine then it is possible to manipulate it directly. The main reason for wanting to do this is speed and so POKEs (and PEEKs) make up the greater part of many moving or dynamic graphics programs.

DYNAMIC GRAPHICS

Moving graphics are the main constituent of almost all computer games. This is not to say that they haven't any 'real' applications it just emphasises how much fun dynamic graphics can be. Although it is often obvious how to draw simple shapes on the screen it is often something of a puzzle to know how to make them move and even more of a puzzle to know how to produce all the dazzling effects seen in computer games — collisions, explosions, bouncing balls etc. While it is important to realise that many of the most stunning effects produced by games machines are the product of special hardware it is easy to produce very good moving graphics even from BASIC.

The principle behind moving graphics is not difficult to understand. A cine film (or a normal television picture) gives the impression of movement by showing a sequence of motionless pictures. The sequence of pictures is in fact a jerky and jumpy approximation to the truly smooth movement that it represents but the eye is fooled into smoothing the jumps out. This perception of a sequence of still images as a smoothly moving object is possibly the only visual illusion that is used by technology. Making things move on a computer screen uses the same principle. The only problem is how to produce the sequence of images required. If you want to make something move across the screen then all you have to do is to plot it then remove it (unplot it) and then plot it again but at a slightly different position. By repeating this plot/unplot/move action the object can be made to 'slide' about the screen.

For example, to make something move horizontally try:

```
10  X=1
20  Y=middle
30  PRINT TAB(X,Y);"A"
40  PRINT TAB(X,Y);"[SPC]"
50  X=X+1
60  GOTO 30
```

For simplicity we are assuming that PRINT TAB(X,Y) will position the cursor at the Xth column and the Yth row. Of course this has to be replaced by whatever graphics command your version of BASIC has. Also 'middle' should be replaced by a value that starts the 'A' off in a reasonable position to be seen.

If you do run the above program on your machine you will see a letter 'A' shoot across the screen but you might not be very pleased at how smoothly it moves. On most machines it will in fact appear to twinkle as it moves. The reason for this is two-fold. Firstly, the letter 'A' moves in 'jumps' that are rather too big to completely fool the eye into seeing smooth motion. And secondly there is an imbalance between the time that the 'A' is displayed and the time that it is not. Ideally the time that the 'A' is on the screen should be long compared to the time that it is off. Unfortunately the above program doesn't really leave the 'A' on the screen for long enough before removing it by printing a blank at the same place. To see the effect of increasing the time that the 'A' is on the screen try adding:

```
35  FOR I=1 TO N
36  NEXT I
```

with different values of N to produce different delays between printing the 'A' and removing it.

This plotting and then unplotting is the way all dynamic graphics are produced. The only trouble is that between the plotting and unplotting you have to do all the calculations concerning the movement of the object (and any other objects being moved by the program). In BASIC this can take so long that dynamic graphics look more like slow motion. There is no solution to this problem apart from moving to a faster language and this often means machine code.

VELOCITY

Making something move across the screen in a straight line as in the last example is all very well but it's hardly inspiring. The next step is to allow the moving object to move in more complicated ways. Although this is just a matter of calculating the positions that the ▶

object should take up it is more useful to think of this in another way.

If the time interval between plotting and unplotting an object is constant (and it nearly always is) then the distance that the object moves can, in some senses, be thought of as being related to a 'velocity'. The higher the velocity the greater the distance moved each time. Putting these ideas into practice involves defining two 'velocities' a horizontal velocity and a vertical velocity. If these are stored in two variables H and V then the process of calculating the new position of the object is simply:

```
X=X+H
Y=Y+V
```

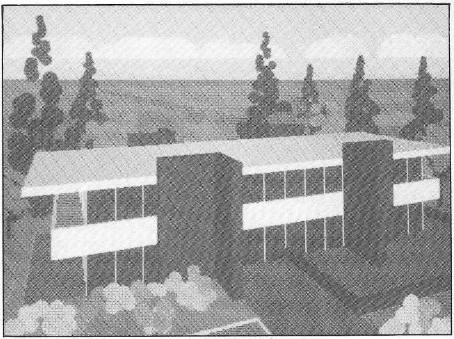
In other words in each time period the horizontal position changes by an amount equal to the horizontal velocity and the vertical distance changes by an amount equal to the vertical velocity. As an example consider the following program using the same conventions as the last example:

```
10  V=1
20  H=1
30  X=1
40  Y=1
50  PRINT TAB(X,Y);"A"
60  PRINT TAB(X,Y);"[SPC]"
70  X=X+H
80  Y=Y+V
90  GOTO 50
```

This makes the letter "A" move diagonally across the screen.

The full advantage of treating the movement of objects in terms of velocities is easy to see once you consider the problem of 'bouncing' a 'ball' around the screen. If the 'ball' meets a vertical 'wall' all you have to do is to reverse the horizontal velocity. If the 'ball' meets a horizontal 'wall' then the vertical velocity has to be reversed. In a cross between English and BASIC this becomes:

```
85  IF 'at horizontal wall' THEN
    V=-V
86  IF 'at vertical wall' THEN H=-H
```


An example of what can be achieved with advanced graphics.

If you can translate the conditions in each of the IF statements into proper BASIC for your machine then these two lines can be added to the previous example to give a program that 'bounces' the letter 'A' around the screen! The same method can be used to 'bounce' objects off bats etc.

ACCELERATION

The idea of using velocities to control the motion of objects is very useful but results in things travelling only in straight lines. To introduce curvature to the paths that objects take it is necessary to add the idea of acceleration. If at each step we not only modify the position of the object but also the velocities then it will travel along curved paths. For example, if we introduce a vertical acceleration we can produce a falling object:

APPLE II
USER'S GUIDE
TRS-80 ASSEMBLY
LANGUAGE
MADE SIMPLE
Program
Pocket Guide

```
 10  H=1
 20  V=0
 30  A=0.2
 40  X=1
 50  Y=1
 60  PRINT TAB(X,Y);"A"
 70  PRINT TAB(X,Y);"[SPC]"
 80  X=X+H
 90  Y=Y+V
100  V=V+A
110  GOTO 60
```

As the object (the letter 'A') moves across the screen the vertical velocity increases because of line 100. This steady increase in velocity causes the 'A' to fall in a parabola. By using horizontal and vertical velocities and accelerations it is possible to produce a wide variety of motions.

CONCLUSION

Dynamic graphics is one of the most enjoyable areas of programming and with very little extra knowledge a wide range of effects can be achieved. Once the basic method of making things move is understood nearly everything else can be handled in terms of velocity and acceleration. To make a number of objects move at the same time all you have to do is keep track of the position, velocity and acceleration of each one of them. In theory this sounds quite straightforward but in practice, of course, it may be quite a confusing task — especially if you ignore the hints contained in earlier parts of this series!

Mike James

ELEGANT PROGRAMMING – 8

Sorting and searching lists can be a difficult matter.

It is something of a shame that the twin topic of sorting and searching sounds as though it is a dull, dry and uninteresting subject because, apart from being something every programmer should know a little about, it is also a subject that contains enough surprising and ingenious programming methods to keep any programmer amused! Perhaps the trouble with sorting and searching is that unless you have tried to do either you will underestimate some of the tricky little problems that crop up. This is not to say that humans don't have very much to do with sorting and searching in everyday life. For example, the results of examinations have to be sorted into order, every time you look up a telephone number or a word in a dictionary you are using a search algorithm — even if you don't know which one! By examining the ways that humans carry out sorting and searching you can begin to write programs that do the same things but to make the best use of the special talents and limitations of a computer then you have to think a little harder about what it is you are trying to do.

THINGS TO SORT

The reasons why lists of things are more useful in some kind of order are numerous. In particular, things are easier to find in a sorted list and herein lies the connection between sorting and searching! It is also true that humans are more at home with lists of information in order and tend to worry about a filing system that is chaotic even if there is a way of finding anything that you might want.

The range of things that constitute lists to be sorted is wide. However, from the computing point of view this range can be reduced to considering sorting a list consisting of two pieces of information. The first is the item that you are basing the order of the list on, it could be a number, a letter or even a name. This is often referred to as the 'sort key' or just the 'key'. The second is a number that indicates where the information that constitutes an element of the list is stored. For example, you may want to sort a list of names and addresses into alphabetical order. The ordering of the list is only on the basis of the name part of the name and address record and hence this is the sort key. As you might imagine sorting such a list involves moving a lot of information around. Instead of having to move an entire record consisting of a name and perhaps a very long address each time it is better to leave the records at fixed locations and move 'pointers' to their locations instead. This idea may seem overcomplicated, especially for BASIC, but in fact it can be implemented very easily. If you assume that the name and address, or any other information to be sorted, is stored as elements of an array (or even a number of arrays) then you can set up another array that holds a list of pointers to the elements of the first array. So if the names and addresses are stored in N$ and the pointers in N the location of the first name and address record is in N(1) and N$(N(1)) is the record itself. In the same way N$(N(2)) is the second record, N$(N(3)) is the third and so on. Sorting the list of names and addresses can obviously be done by moving only the elements of the array N and leaving all the elements of N$ exactly where they were. The saving in time gained by using the second array N to hold the order of N$ far outweighs the cost of the extra memory needed for N.

HOW WELL DOES IT SORT?

If you have a program that sorts a list into order then obviously it is important that it does its job fast. There are some sorting algorithms that work well enough with small amounts of data but the time that they take increases rapidly if you increase the amount of data to be sorted. It is important to know how well a sorting method works but how do you measure this? The time that most sorting methods takes depends on the actual set of data you give it to sort as well as how much of it there is. For example, if the list happens already to be in order then nearly all sorting methods will notice this and stop after doing very little work. A similar speed increase is the case in less extreme examples, in other words the more the list is already in order the faster a sorting method works.

COMPARING THE SORTS

An obvious measure of how good a sorting method is, is how fast on average it will sort a list of a particular size. There will be lists that are sorted faster and there will be lists that take longer. In fact there will be lists that take a lot longer than the average time — for example, a list that is ordered in entirely the reverse order to that desired, such lists are often called 'pathalogical'. When comparing sorting methods there are two things to be taken into account — the average time to sort a list and the longest time to sort a list.

For instance, if you find a sorting method that is fast on average but incredibly slow on a pathalogical list then make ▶

sure all your lists are well behaved or be prepared for a long wait every now and again! Obviously the time that an ideal sorting method takes should increase as slowly as possible as the length of the list increases, should be fast on average and not much worse for a pathalogical list. Now that we know what we want it has to be admitted that most sorting methods are far from ideal!

SOME SIMPLE SORTING METHODS

If you were given a pile of marked exam papers and were asked to put them in order you might proceed as follows:

> 1) scan through all the papers and find the one with the highest mark;
> 2) place this paper at the front of the pile.
> This paper is now 'sorted' and takes no part in any further sorting;
> 3) repeat (1) and (2) with the 'unsorted' remainder of the pile, putting the highest one in front of the remainder, until all the papers are sorted.

This is known as a 'selection sort' because it is the repeated application of selecting the highest mark from the remaining papers that is used to sort the pile. In BASIC this is:

```
10 LET N=10
20 DIM L(N)
30 GOSUB 1000
40 GOSUB 2000
50 GOSUB 3000
60 STOP

1000 FOR I=1 TO N
1010 LET L(I)=RND
1020 NEXT I
1030 RETURN

2000 FOR I=1 TO N-1
2010 LET M=I
2020 FOR J=I+1 TO N
2030 IF L(M)<L(J) THEN LET M=J
2040 NEXT J
2050 LET T=L(I)
2060 LET L(I)=L(M)
2070 LET L(M)=T
2080 NEXT I
2090 RETURN

3000 FOR I=1 TO N
3010 PRINT L(I)
3020 NEXT I
3030 RETURN
```

Subroutine 1000 constructs a list of random numbers to be sorted and subroutine 3000 prints the list. Both of these subroutines will be used in the following examples without being given again. Subroutine 2000 is the actual sort part of the program. It works in much the same way as the 'human' method except it uses a single array rather than a 'pile' of papers. Lines 2010-2040 find the largest value in the array starting from L(I) and ending with L(10). Lines 2050-2070 carry out a 'swap' between L(I) and the largest element. Finding the largest value and swapping it is repeated until the array is sorted by the FOR loop starting at line 2000.

The selection sort is not a fast method. Just by looking at the way the two FOR loops are nested you should be able to see that if L has N elements the complete sort will take roughly N *N/2 comparisons and 3 *N swaps. This figure, however, is not affected at all by any order already present in the list so the selection sort is equally bad for all cases!

Another sorting method based on the way a human might do it is the 'insertion' sort. If you were given a hand of cards to arrange in order you might proceed something like:

a) Place the first two cards in their correct order;
b) Then place the third card correctly within the first two;
c) Then place the fourth card in its correct place within the first three and continue like this until all the cards have been sorted.

This idea is easy to convert into BASIC by using two arrays. The first holds the items to be sorted and the second is used to place the items one at a time into order.

```
10 LET N=10
20 DIM L(1N)
30 DIM A(N)
40 GOSUB 1000
50 GOSUB 2000
60 GOSUB 3000
70 STOP

2000 LET A(1)=L(1)
2010 FOR I=2 TO N
2020 LET J=1
2030 IF L(I)<=A(J) THEN GOTO 2090
2040 FOR K=I-1 TO J STEP -1
2050 LET A(K+1)=A(K)
2060 NEXT K
2070 LET A(J)=L(I)
2080 GOTO 2130
2090 LET J=J+1
2100 IF J<=I-1 THEN GOTO 2030
2120 LET A(J+1)=L(I)
2130 NEXT I
2140 RETURN
```

and change line 3010 to:

```
3010 PRINT L(I),A(I)
```

The sort is once again carried out by subroutine 2000. Line 2000 takes the first item in the list L and places it in the array that the sorted list will be stored in (ie A). The FOR loop starting at 2010 is responsible for inserting each of the items from L into A. Lines 2020-2120 are responsible for actually carrying out the swap. The new item is compared with the items already sorted in the array A by line 2030. When the correct position for the new item is found then lines 2040-2060 make space for it by moving all the elements that are smaller than it up the array by one place. Then the new item is inserted by line 2070. If the new item is smaller than any already in the array A it is just appended to the end by line 2120 without having to make any space.

This procedure looks as though it will be slower than a selection sort because of all the moving of already sorted items to make room for new items. However, things are not what they seem. The number of comparisons needed to sort N items is only N *N/4 which is better than selection sort. The number of moves of items is also N *N/4 which is better than the 3N swaps of selection sort except when N is tiny.

OTHER SORTS

The next sorting method, the 'bubble sort', is perhaps a little too well known. This is really the first method that is not based on the way a human would sort things but tries to make use of the speed with which a computer can do simple things. The basic bubble sort algorithm is easy to understand. All you have to do is make a 'scan' through the array comparing items that are 'next door' to each other. If they are in the correct order then nothing is done. If they are in the wrong order then they are swapped. This scan is repeated until the list is fully sorted and there are no swaps made on a scan. The BASIC for this is:

```
10 LET N=10
20 DIM L(N)
30 GOSUB 1000
40 GOSUB 2000
50 GOSUB 3000
60 STOP

2000 LET F=0
2010 GOSUB 2500
2020 IF F=0 THEN RETURN
2030 GOTO 2000

2500 FOR I=1 TO N-1
2510 IF L(I)>L(I+1) THEN GOTO 2560
2520 LET T=L(I)
2530 LET L(I)=L(I+1)
2540 LET L(I+1)=T
2550 LET F=1
2560 NEXT I
2570 RETURN
```

The structure of this program is very simple to understand. Subroutine 2000 first sets a flag F to zero and calls subroutine 2500 which performs a scan and sets the flag F to 1 if a swap occurs. Line 2020 repeats the scan until the F is zero indicating that the array has been completely sorted. Subroutine 2500 performs the scan using a FOR loop and compares L(I) with L(I+1). The swap is carried out by lines 2520-2540 and the flag is set in 2550.

It is a fact that the bubble sort is so easy to understand and quick to program because it is far from a good method of sorting. You need about N *N/2 comparisons and 3 *N *N/4 swaps to sort a list of N items and this is worse than an insertion sort! The interesting thing about the bubble sort is that you can find lists that it works very quickly on — for example, a list that has only a few well separated pairs of items in the wrong order will be sorted in one scan — but there are also lists that it does very badly on — for example, a list in reverse order. In conclusion, the bubble sort is slow and very sensitive to the initial order of the list.

There is an improvement to the simple bubble sort that makes it very good indeed. The main trouble with the bubble sort comes from the fact that items move by only one place at a time. If an item is at the bottom of the list and it should be at the top then it will take N scans to move it to its correct place. If we extend the bubble method to comparing and swapping items that are further apart than one then we might improve the speed with which items reach their final resting place. If the distance between items compared and possibly swapped is D we call the resulting sort a D-sort. In other words, the bubble sort can also be called a 1-sort. The only complication is that a list can be sorted as far as a 4-sort, say, is concerned but still not sorted as far as a 1-sort is concerned. In practice we employ a sequence of sorts with decreasing distance between the items compared until a 1-sort finally gives the fully sorted array. This method is often called a Shell sort after the person who first invented it. A Shell sort in BASIC is not much more difficult than a bubble sort:

```
10 LET N=10
20 DIM L(N)
30 GOSUB 1000
40 GOSUB 2000
50 GOSUB 3000
60 STOP

2000 LET D=8
2010 IF D<1 THEN RETURN
2020 LET F=0
2030 GOSUB 2500
2040 IF F=1 THEN GOTO 2020
2050 LET D=D/2
2060 GOTO 2010

2500 FOR I=1 TO N-D
2510 IF L(I)>L(I+D) THEN GOTO 2560
2520 LET T=L(I)
2530 LET L(I)=L(I+D)
2540 LET L(I+D)=T
2550 LET F=1
2560 NEXT I
2570 RETURN
```

The way that this program works can be understood by comparing it with the bubble sort given earlier. The distance between items compared is stored in D and starts out at 8. After each scan at a particular D returns without making any swaps, the value of D is halved and a new series of scans is initiated. This continues until a 1-sort returns without making any swaps. Thus the sequence of sorts is 8-sort, 4-sort, 2-sort, 1-sort.

It is difficult to say how good the Shell sort is on average as it depends on the initial value given to D but the main point is that it is a good sorting method to use unless you are going to be sorting very large lists of data.

This examination of sorting methods could continue without end. There are better methods of sorting data than the Shell sort but these would need an article each to explain. The Shell sort is one of the simplest efficient sorting methods. There is never a good reason for using a bubble sort. If you need to sort very large lists of data then you will need a method that is even better than the Shell sort. Some names to look to up in reference books include tree sort, heap sort and quick sort.

RUNNING OUT OF MEMORY

Each of the sorting methods described above only work if all the data can be stored in memory at once. What do you do if you have a very large disc file to sort? The answer is surprisingly simple and leads to a sorting method all on its own.

Let's suppose that you have a list of 100 records to sort and only enough memory to hold 50 of them at a time. All you have to do is to split the list into two parts of 50 records each and sort them independently of one another in memory. To make a single fully sorted list of 100 records all you have to do is to merge the two files in the following way:

1) read item1 from file1 and item2 from file2;
2) write the largest of item1 and item2 to output file;
3) read a new item from correct file to replace item written out;
4) repeat (2) and (3) until both files are empty.

As an example, consider merging the two lists 9,5,3,1 and 8,7,6,2. The first two items are 9 and 8, and as 9 is greater than 8, 9 is written out. As the item from list one was written out, the next pair of items is 5 and 8 and, as 8 is greater than 5, 8 is written out and the next item is read from list two. If you carry on you will eventually get a fully sorted list of eight items. ▶

In general, to sort a list too big to be held in memory simply split it up into a number of smaller files that will fit into memory and then sort these. To obtain the final sorted list all you have to do is perform a merge on all the smaller sorted files, reading one item from each, writing out the largest and replacing it by a new item.

You can use merging as a sorting method by dividing up the list of items into files consisting of one item! These one element lists are obviously sorted and can be combined together in pairs using merging. The resulting two element files can then be merged in pairs to yield sorted files of four items and so on.

Merging is important for a number of reasons — it is useful to know how to combine two or more sorted files, it can be used as a sorting method in its own right and it allows lists of data too big to be sorted in memory to be sorted in sections.

SEARCHING

The time has come to consider our other topic this month — searching. In the same way that the data item that a list is ordered on is called the 'sort key', the data item that is used to detect a match during a search is called a 'search key'. For example, if you have a list of names and addresses and you are searching for someone with a particular name then the name is the search key.

If the list that you are searching isn't sorted then there is really no alternative to starting at the top and comparing the search key with every item. If the list has N elements then on average this process will take N/2 comparisons to come up with an answer. However, if the list is ordered then there is a much better method of searching it called binary search. If we assume that the list is in ascending order, with the largest value at the bottom of the list, then binary search works like this:

1) starting at the middle of the list,
   if the middle item is bigger than the search key then the item we are looking for must be in the upper half of the list;
   if the middle item is smaller then the item we are looking for is in the lower half of the list.
2) repeat (1) on the half of the list that the data item lies in until it is found.

For example, if the list is 1,3,5,7,8,9,12 and we are looking for 3, the search begins at the middle of the list ie the fourth item. This is 7 which is bigger than 3. This tells us that what we are looking for is in the first half of the list. Continuing the search at the middle of the first half we examine the second item or 3 which is of course what we are looking for. A binary search takes on average only logN comparisons which is considerably faster than N for the direct search.

MAKING A HASH OF IT?

If you have a sorted list to search then you cannot do much better than use a binary search. The question is what do you do with a list that is being built up and searched continuously. You could spend a long time sorting it before every search. Anyway for very large lists sorting can take a very long time. The answer to these problems is that if you can afford more storage than strictly needed for the list you can use a technique known as 'hash coding'. (Notice that this is another example of the general principle of trading memory for speed.) Hash coding works like this:

1) when an item is added to the list its search key is used to calculate the place in the list where it should be stored. The equation used to calculate this is known as the 'hash function'.
2) when you need to find an item the hash function is again applied to the sort key and this produces the location where the record can be found without any searching for it at all!

For example, consider a list of names and addresses that have to be searched for a particular name. A possible hash function is to convert each of the letters of the name to its ASCII code and add them up to produce a single number. if we assume that there are 10 letters in a name the result of this hash function will lie in the range 0 to 2560. This could be used as the index to any array that was used to store the information. So when a new name and address arrives we apply the hash function to the name to get a number that is used as an index to the element of the array that the name and address will be stored in. To find the address corresponding to a name we once again apply the hash function to produce a number that is the index of the element where the name and address is stored.

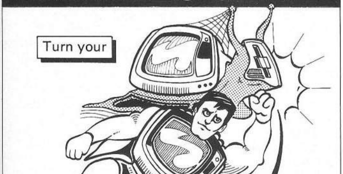
This use of a hash function sounds rather like magic. After all it allows you to store and retrieve data without any searching! The difficulty with this method is selecting a good hash function. In practice, any hash function will return the same suggested location for a number of values of the search key. In the previous example, where the hash function consisted of converting the letters to ASCII and adding them together, it should be obvious that the name 'SMITH' and the (unlikely but possible) name 'MITHS' will both give the same result. If this happens where should the records be stored? The usual solution to this 'collision' problem, as it is known, is to store the first record in the location given by the hash function but store all subsequent entries in the first free location following its position as allocated by the hash function. To find records when there is the possibility of collision we have to modify the hash technique to include a search. The hash function is applied to the key. If this gives the location of the required record then fine, if not you have to examine the subsequent locations in the list until you find what you are looking for.

As a result of having to handle possible collisions you need to insure that there is some spare space in the array to accommodate the 'displaced' records. This is the reason that you have to use an array that is strictly larger than the number of records that you want to store. The more spare space in the array the faster the search will be. It is the collision problem that makes hash coding not quite as fast as it might be, but with a good choice of hash function (ie one that produces as few collisions as possible) it can be remarkably good. It can often take only one or two comparisons to find the record you are looking for even in very large lists of data as long as the array that is used to hold them is reasonably empty. In fact, even the condition that the array be reasonably empty is not too troublesome. For example, with a good hash function and a table 95% full the search only goes up to 4 or 5 search key comparisons!

The subject of hash coding and choosing a hash function is too big to go into in any more detail here. There are plenty of reference books that cover the subject in great detail however! If you have followed the basic idea you should be able to see why it is worth looking hash coding up.

Sorting things out is not always easy!

TRS80 Models I+III and VIDEO GENIE



ACCEL3 – the practical BASIC compiler for home, education, or business.

Are you troubled by gradual graphics, languid loops, tedious table searches, or capricious keyboard response? ACCEL3 is the cure. Highly compatible with interpreted BASIC – correct programs compile without modification.

On Tape or Disk £49.95

southern software

PO Box 39, Eastleigh, Hants, SO5 5WQ

BRANE SOFTWARE

SCROLLER (16/48K) Enhance your programs with enlarged sideways scrolling messages.Height magnification up to 8 times width magnification up to 32 times.Full choice of colours, size and position of window ,etc.........£7.95 incl p&p

ADVERTISER(48K) Thought of buying a message scrolling display for eye catching 24 hour advertising? This program is more versatile and much cheaper!.Complete set of up to 40 different displays of almost any length.........£17.95 incl p&p.

CUT YOUR HEATING BILLS (48K) Your spectrum can save you more than it cost to buy!No technical knowledge required £7.95 incl p&p.A system that takes the work out of estimating the savings made by insulation/double glazing.

All programs menu driven,fully error trapped and guaranteed Trade enquiries welcome:

Brane Software,Myrtle Grove,Brane,Sancreed,Penzance,
TR20 8RE
Tel: 073 672 562

007 SPY KOPYKAT

Simply load a program into your Spectrum and press 'C' for instant tape copy. Works on Basic, Screen★, Machine-Code (bytes) and Headerless Datafiles. This is a genuine copier and does **NOT** use any of your user memory.

Includes a free program to let you stop and look at virtually any Basic/Bytes programs. And can be used to copy from tapes to microdrive.

£3.95

G. J. BOBKER
29 CHADDERTON DRIVE
UNSWORTH, BURY,
LANCS BL9 8NL

Mike James

ELEGANT PROGRAMMING-9

For our grand finale in this series we take a look at a completely different method of tackling problems in programming.

he subject of this final article is the rather grand sounding programming technique called, 'recursion'. You may feel that the last part of a series on programming is an odd place to be introducing yet another programming method! After all, stepwise refinement and structured programming were covered earlier in the series as the only way to write good programs — so what else can there be? The answer is that recursion is almost a wholly separate approach to programming. Simple programming problems are most easily solved using combinations of branch (ie IF statements) and loops. Problems that are best solved using recursion are usually not encountered until much later on the road to becoming an expert programmer and, by this stage, it is often too late to see an alternative way of tackling a problem. This might account in part at least for the trouble that many people have with understanding and using recursion. On the other hand it might just be that recursion is a method of thinking that you either find natural or you don't. Whatever the reason, recursion has a way of making fanatical friends and devoted enemies. A more balanced view is that recursion is just another weapon to be added to the programmers' arsenal and used when appropriate. So, if you have never met recursion or if you have been convinced that it is a difficult technique reserved for academics then read on.

ITERATION V RECURSION

The best way to explain recursion is by example. Perhaps the most used and simplest example of recursion is the calculation of the factorial function. It is a good example not only because it is simple but because it shows clearly the relationship between the programming methods we already know — looping etc — and recursion. The factorial function n! is the product of all the integers from 1 up to and including n. In other words:

$$n! = 1*2*3*4*5\ldots*(n-1)*n$$

If you were set the task of writing a BASIC program to calculate the factorial function then you would probably write something like:

```
10 INPUT N
20 A=1
30 FOR I=1 TO N
40 A=A*I
50 NEXT I
60 PRINT A
```

The main part of the program, ie the part that does all the work is the FOR loop between lines 30 to 50. The usual name for this sort of solution is 'iteration'. Any program that arrives at its solution by going round a loop is known as an iterative program. At this point it may be difficult to see how there could possibly be an alternative to iteration — looping is so fundamental to programming. However there is another equivalent definition of the factorial function that leads directly to a different sort of program that calculates it. If you want to work out n! and you happen to already know what $(n-1)!$ is then you can take a short cut by using:

$$n! = n\star(n-1)!$$

For example $4! = 4\star3!$. If you don't happen to know the value of $(n-1)!$ then you can use the same idea once more to find $(n-1)!$ That is $(n-1)! = (n-1)\star(n-2)!$ You should be able to see that you can keep on using this relationship until you get to a factorial that you do know the value of and then by working your way back up the chain you can return to the value of the factorial that you want. A value of the factorial function that is particularly easy to remember (or work out) is 1! which is of course equal to 1. So, for example, to calculate 4! using this method we would first reduce the problem to finding 3! by $4! = 4\star3!$. Then we would reduce the problem to finding 2! by $3! = 3\star2!$ and finally to 1!, which we know by $2! = 2\star1!$. To get the required answer we now have to work our way back 'up' the chain of calculations ie $2! = 2\star1! = 2$, $3! = 3\star2 = 6$, and finally $4! = 4\star6 = 24$. This, rather strange method calculates the factorical function without any hint of an iterative loop — it is the recursive method of calculating the factorial function. The ideas of 'stepping down' through a calculation until you reach a point where you can replace unknown parameters by actual values and then 'stepping up' through the calculation filling in the previously missing values is characteristic of all recursion. Another feature of recursion is the way that the recursive definition of the factorial function involves itself. That is:

$$n! = n\star(n-1)!$$

can be read as a definition of n! in terms of n and $(n-1)!$ In fact it is this self referencing that makes the step down/step up behaviour of recursion possible.

Now that we have an alternative method of calculating the factorial function the next step is to produce a BASIC program that uses recursion. However this is not quite so easy as it sounds.

RECURSION AND BASIC

There are computer languages

that are defined and implemented with special features to allow and even encourage programmers to write recursive programs. The trouble is that BASIC isn't one of them! This isn't unreasonable when you think of BASIC's humble origins as a first teaching language. A few versions of BASIC — C BASIC and BBC BASIC for example — contain special facilities for recursion but, in general, BASIC leaves the programmer to sort out recursion alone. Things are not quite so bad, however, because it is fairly easy to write clean and neat recursive programs in BASIC using a simple idea. Before introducing this it is worth looking at the way that a standard recursive program would appear in a version of BASIC that facilitates recursion — BBC BASIC.

What we need to do is take the recursive definition of the factorial function and convert it as directly as possible into BASIC:

```
10 DEF FNF(N)
20 IF N<>1 THEN =N*FNF(N-1)
30 =1
```

Although the above program may look a little strange you should be able to identify the overall form of a recursive subroutine. The first line (10) defines what follows as a function called FNF. The second line (20) is the recursive definition of N! It says IF N< >1 THEN the result of the function is N times the result of FNF(N− 1). You should be able to see that this is where the step down/step up calculation occurs. When FNF is used, for example in the statement PRINT FNF(4), line 20 causes FNF to be called as FNF(N− 1), FNF(N− 2) and so on until FNF(1) is reached when line 30 returns the value 1 and the chain of calculations is taken back up towards the first use of FNF. This idea can only work and really be understood if each time FNF is used a completely new version of the function, in particular all its variables, are created anew. For example in the execution of FNF(4) line 20 causes a completely new version of FNF to come into existence to work out FNF(3). This in turn causes another version of FNF to come into existence to work out FNF(2) which finally creates a version to give the value of FNF(1). Not only must a new version of FNF come into existence each time it is used, each new version must only replace the previous one until it returns a result. In other words to allow the calculation to work its way back up the chain it is necessary for each of the versions of FNF created on the way down the chain to carry on existing both to accept the results of the later versions of the function and to return a value to any earlier versions of the function. So to continue the above example, when FNF(1) returns the value 1 as its answer, it passes it to the partially completed version calculating FNF(2). This allows this version to complete its line 20 and pass the result 2 to the next version and so on to the first use of FNF which finally returns the value FNF(N) to the PRINT statement that it was used in. The way that FNF works out any factorial is not difficult to understand but it may be difficult to follow so Fig. 1 is included as a summary of the FNF(4) calculation.

This description of the FNF function is all very well for anyone with BBC BASIC but what about the rest of us. Well the answer is that there is a simple method of implementing recursive subroutines in almost any version of BASIC. The method relies on the version of BASIC having a good pair of GOSUB and RETURN statements. In particular it is important that you can GOSUB to a subroutine from within a subroutine and still have the RETURN statement take you back to the correct place. In other words it is important that subroutine calls can be 'nested' to a reasonable depth. The main problem in using BASIC subroutines recursively is that each time the subroutine is used a whole new set of variables should come into existence and when the subroutine finishes it should be possible to return to a previous version of the subroutine restoring the old values. The simplest, but incorrect, BASIC recursive implementation of the factorial function is:

```
10 INPUT N
20 GOSUB 1000
30 PRINT F
40 END

1000 IF N=1 THEN F=1:RETURN
1010 N=N-1
1020 GOSUB 1000
1030 F=(N+1)*F
1040 RETURN
```

Subroutine 1000 attempts to use recursion to calculate N! by calling itself at line 1020 to work out an answer for (N− 1)! and then using this in line 1030 to calculate N!. Unfortunately, this

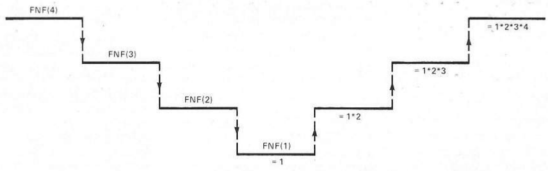


Fig. 1. The calculation of FNF(4).

doesn't work because the old values of N and F are destroyed each time the subroutine is called. The answer to this is to use an array for each variable in the subroutine and a count of how many times the subroutine has been called. This count is used as an index to the arrays so that effectively a completely new set of variables is produced each time the subroutine is called. For example:

```
10 DIM N(10)
20 DIM F(10)
30 INPUT N(1)
40 I=0
50 GOSUB 1000
60 PRINT F(1)
70 END

1000 I=I+1
1010 IF N(I)=1 THEN F(I)=1:I=I-1:RETURN
1020 N(I+1)=N(I)-1
1030 GOSUB 1000
1040 F(I)=N(I)*F(I+1)
1050 I=I-1
1060 RETURN
```

The two simple variables N and F are now replaced by arrays N(10) and F(10). The variable I counts the number of times that the subroutine is called. Within the subroutine the current values of the variables are in N(I) and F(I) respectively but the result from the previous version of the subroutine is always in $F(I+1)$ and the value of N is passed in $N(I+1)$. This is how the versions of the subroutine communicate with each other.

This use of arrays to create new versions of the variables each time the subroutine is used is interesting because it imitates the way that languages such as Pascal implement recursion. You may recognise the way the arrays are used with the index I as nothing more than a simple stack.

RECURSIVE SOLUTIONS

The recursive calculation of the factorial function is a good example because it is easy to see how the definition leads to the program. Recursive programs do often arise directly from the implementation of a recursive definition but it is also the case that many problems that seem to have nothing to do with recursion at first sight can be solved by recursion. For example the well-known 'Towers of Hanoi' problem contains no obvious hint of recursion but it is most easily solved by recursion. The Towers of Hanoi problem consists of three pegs numbered 0, 1, and 2 and at the start of the problem there is a pyramid of N discs, smallest at the top on peg 0. The object of the puzzle is to transfer all the discs one at a time to another peg but with the restriction that a larger disc must never be placed on top of a smaller disc. If you have never come across the problem before, you may not appreciate just how tricky it is. Try it for yourself using four or five coins and you'll soon understand the difficulties. A recursive solution consists of four stages:

1) If $N=1$ then move the disc from peg 0 to peg 1 and stop

2) If N 1 then move the top $N-1$ discs to peg 2

3) Then move the remaining bottom disc to peg 1

4) Move the $N-1$ discs now on peg 2 to peg 1

Steps 2 and 4 are clearly recursive in that they both involve the original problem but with $N-1$ discs instead of N. Uncovering this sort of solution is something that gets easier with practice but the main idea is to reduce the problem you are faced with to a solution of a slightly simpler one and then repeat this reduction until the problem is solved.

SUMMING UP

Recursion is a subject that has received much academic attention. Rather than just being an alternative to iteration it may be that recursion is in some way more powerful. In other words there may exist problems that cannot be solved using iteration but can using recursion. More to the point, it is possible that there are practical problems that are significantly easier to solve using recursion. For example, most compilers analyse computer languages using recursive methods. Many of the problems in artificial intelligence seem to be easier to understand and solve using recursive methods. Whatever the truth, recursion is finding its way into computer languages intended for advanced future applications.

On the subject of programming languages the choice of BASIC as the language for examples in this series may seem a little strange — if you want to illustrate advanced or good programming methods and ideas then surely an advanced language would be the best choice. Apart from the obvious advantage of using the most popular and common programming language (ie BASIC) it also serves to emphasise the fact that the techniques are ideas independent of any particular language. BASIC is by no means the last word in programming languages but then neither is the much praised Pascal. It is true that any language that provides extra facilities for writing well structured programs with advanced data types and structures is to be admired but it is always up to the programmer to take advantage of these facilities. I have seen as many badly written programs in Pascal as in BASIC! Programming languages will develop and offer more advanced features as time goes on but it will take a lot to move BASIC from its current position as the number one language. What is likely to happen is that BASIC will develop to include extra features until it becomes nothing like the BASIC that we use today.

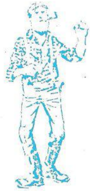
This aspect of evolution rather than revolution in programming languages is very like the way natural languages develop and why should it be otherwise? As long as the more advanced versions of BASIC include the original as a subset there should be no problems. Until the day that computers program themselves we must continue to find ways of improving the clarity and accuracy of the programs that we write and this will entail the further development and refinement of high level computer languages, BASIC included.

YEP FOLKS — IT'S HERE

AVAILABLE NOW
Spectrum 48K
Dragon
Com. 64

CALIFORNIA

GOLD RUSH



HOWDE DO PARDNERS
This here's Prospector Jake, I sure am havin' one helluva time tryin' to peg ma claim with those damned Injuns a hootin' an a hollerin' all over this territory. Ma job gets harder as I move from one Gold Field to another. I know, that is me an' ma stubborn hornery ol' Mule here know of 24 rich an' I mean rich seams of pure Gold. All it needs to make this here ol' critter happy is that you help me peg every doggone last one of them claims.

Can YOU help Jake become rich, help him peg his claim, dodge the arrows, avoid the tomahawks, and plant the Dynamite in just the right place? . . . YOU CAN!!!

YIPPEE . . . Git yer Picks an' Shovels and join the CALIFORNIA GOLD RUSH . . . NOW

Amazing Arcade Action . . . Stunning Sound and Graphics
Available NOW for Commodore 64, Spectrum 48, and Dragon

£ 7.95 including P&P

SPECIAL OFFER **SPECIAL OFFER** **SPECIAL OFFER**

Order CALIFORNIA GOLD RUSH before November 11 and get a 10-game Cassette of terrific games . . . **FREE**

COMING SOON

LEAPIN' LANCELOT: Medieval Machine Magic to enthral you
GALACTIC SURVIVAL PAK: Every Astro-Traveller *must* have this!

ANiK

We always need Dynamic Dealers and Imaginative Writers

Please rush me CGR for (m/c)

TOTAL SUM INCLUDED £
Please make cheques and POs
payable to ANIK MICROSYSTEMS
30 KINGSCROFT COURT
BELLINGE, NORTHAMPTON

Name.................................
Address.............................
....
....

100 FREE PROGRAMS

FROM SILICA SHOP — WITH EVERY PURCHASE OF AN

ATARI 400 800

ATARI

1978
1979
1980
ATARI 800
400

ATARI PRICES REDUCED!
We at Silica Shop are pleased to announce some fantastic reductions in the prices of the Atari 400/800 personal computers. We believe that the Atari at its new price will become the U.K.'s most popular personal computer and have therefore set up the Silica Atari Users Club. This club already has a library of over 500 programs and with your purchase of a 400 or 800 computer we will give you the first 100 free of charge. There are also over 350 professionally written games and utility programs, some are listed below. Complete the reply coupon and we'll send you full details. Alternatively give us a ring on 01-301 1111 or 01-309 1111.

ATARI 400 **with 16K**	**£99**
ATARI 400 **with 48K**	**£158**
ATARI 800 **with 48K**	**£249**

400/800 SOFTWARE & PERIPHERALS

Don't buy a T.V. game! Buy an Atari 400 personal computer and a game cartridge and that's all you'll need. Later on you can buy the Basic Programming cartridge (£35) and try your hand at programming using the easy to learn BASIC language. Or if you are interested in business applications, you can buy the Atari 800 + Disk Drive + Printer together with a selection of business packages.

Silica Shop have put together a full catalogue and price list giving details of all the peripherals as well as the extensive range of software that is now available for the Atari 400/800. The Atari is now one of the best supported personal computers. Send NOW for Silica Shop's catalogue and price list as well as details on our users club.

THE FOLLOWING IS JUST A SMALL SELECTION FROM THE RANGE OF ITEMS AVAILABLE:

ACCESSORIES
Cables
Cassettes
Diskettes
Joysticks
Le Stick - Joystick
Misc Supplies
Paddles

ADVENTURE INT
Scott Adams Adv
No 1 Adventureland
No 2 Pirate Adv
No 3 Mission Imp
No 4 Voodoo Cast
No 5 The Count
No 6 Strange Ody
No 7 Mystery Fun
No 8 Pyramid of D
No 9 Ghost Town
No 10 Sav Island 1
No 11 Sav Island 2
No 12 Golden Voy
Angle Worms
Deflections
Galactic Empire
Galactic Trader
Lunar Lander
Mountain Shoot
Rearguard
Star Flite
Sunday Golf

AUTOMATED SIMULATIONS
Crush Crumble Cmp
Datestones of Ryn
Dragons Eye
Invasion Orion
Rescue at Rigel
Ricochet
Star Warrior
Temple of Apshai
Upper Reaches Aps

BOOKS
Basic Ref Manual
Compute Atari DOS
Compute Bk Atari
Compute Magazine
De Re Atari
DOS Utilities List
DOS2 Manual
Misc Atari Books
Op System Listing
Wiley Manual

BUSINESS
Calculator
Database Managemt
Decision Maker
Graph-It
Invoicing
Librarian
Mort & Loan Anal
Nominal Ledger
Payroll
Personal Finl Mgmt
Purchase Ledger
Sales Ledger
Statistics 1
Stock Control
Telelink 1
Visicalc
Weekly Planner
Word Processor

CRYSTALWARE
Beneath The Pyram
Fantasyland 2041
Galactic Quest
House Of Usher
Sands Of Mars
Waterloo
World War III

DYNACOMP
Alpha Fighter
Chompelo
Crystals
Forest Fire
Intruder Alert
Monarch
Moonprobe
Moving Maze
Nominoes Jigsaw
Rings of The Emp
Space Tilt
Space Trap
Stud Poker
Triple Blockade

EDUCATION from APX
Algicalc
Atlas of Canada
Cubbyholes
Elementary Biology
Frogmaster
Hickory Dickory
Inst Comptg Dem
Lemonade
Letterman
Mapware
Maths-Tac-Toe
Metric & Prob Solvg
Mugwump
Music Terms/Notatn
Musical Computer
My First Alphabet
Number Blast
Polycalc
Presidents Of U.S.
Quiz Master
Starware
Stereo 3D Graphics
Three R Math Sys
Video Math Flash
Wordmaker

EDUCATION from ATARI
Conv French
Conv German
Conv Italian
Conv Spanish
Energy Czar
European C & Caps
Hangman
Invit To Prog 1/2/3
Kingdom
Music Composer
Scram
States & Capitals
Touch Typing

EMI SOFTWARE
British Heritage
Cribbage/Dominoes
Darts
European Scene Jig
Hickory Dickory
Humpty Dumpty
Jumbo Jet Lander
Snooker & Billiards
Submarine Commdr
Super Cubes & Tilt
Tournament Pool

ENTERTAINMENT from APX
Alien Egg
Anthill
Attank
Avalanche
Babel
Blackjack Casino
Block Buster
Block 'Em
Bumper Pool
Castle
Centurion
Checker King
Chinese Puzzle
Codecracker
Comedy Diskette
Dice Poker
Dog Daze
Domination
Downhill
Eastern Front
Galahad & Holy Grl
Graphics/Sound
Jax-O
Jukebox
Lookahead
Memory Match
Midas Touch
Minotaur
Outlaw/Howitzer
Preschool Games
Pro Bowling
Pushover
Rabbotz
Reversi II
Salmon Run
747 Landing Simul
Seven Card Stud
Sleazy Adventure
Solitaire
Space Chase
Space Trek
Sultans Palace
Tact Trek
Terry
Wizards Gold
Wizards Revenge

ENTERTAINMENT from ATARI
Asteroids
Basketball
Blackjack
Centipede
Chess
Entertainment Kit
Missile Command
Pac Man
Space Invaders
Star Raiders
Super Breakout
Video Easel

ON LINE SYSTEMS
Crossfire
Frogger
Jawbreaker
Mission Asteroid
Mouskattack
Threshold
Ulysses/Golden Fl
Wizard & Princess

PERIPHERALS
Centronics Printers
Disk Drive
Epsom Printers
Program Recorder
RS232 Interface
Thermal Printer
16K Memory RAM
32K Memory RAM

PERSONAL INT from APX
Adv Music System
Banner Generator
Blackjack Tutor
Going To The Dogs
Keyboard Organ
Morse Code Tutor
Personal Fitness Prg
Player Piano
Sketchpad

PROGRAMMING AIDS from Atari
Assembler Editor
Dsembler (APX)
Microsoft Basic
Pascal (APX)
Pilot (Consumer)
Pilot (Educator)
Programming Kit

SANTA CRUZ
Basics of Animation
Bobs Business
Display Lists
Graphics Machine
Kids 1 & 2
Horizontal Scrolling
Master Memory Map
Mini Word Processor
Page Flipping
Player Missile Gr
Player Piano
Sounds
Vertical Scrolling

SILICA CLUB
Over 500 programs
write for details

FOR FREE BROCHURES—TEL: 01-301 1111

FREE LITERATURE

I am interested in purchasing an Atari 400/800 computer and would like to receive copies of your brochure and test reports as well as your price list covering all of the available Hardware and Software.

Name..

Address..

....

....

....

Postcode..

PS1

SILICA SHOP

For free brochure and reviews on our range of electronic products, please telephone 01-301 1111. To order by telephone just quote your name, address, credit card number, and order requirements and leave the rest to us. Post and packing is FREE OF CHARGE in the UK. Express 24 hour delivery available at an additional charge.

- SHOP DEMONSTRATION FACILITIES – we provide full facilities at our shop in Sidcup, Monday to Saturday 9am to 5.30pm (closing Thursday 1pm, Friday 8pm)
- MAIL ORDER – we are a specialist mail order company and are able to supply goods direct to your door
- MONEY BACK UNDERTAKING – if you are totally unsatisfied with your purchase you may return it to us within 15 days. On receipt of the goods in satisfactory condition we will give you a full refund
- PART EXCHANGE SECOND HAND MACHINES – we offer a part exchange scheme to trade in many makes of T.V. games for personal computers
- COMPETITIVE PRICES – our prices, offers and service are very competitive. We are never knowingly undersold, and will normally match any lower price quoted by our competitors
- HELPFUL ADVICE – available on the suitability of various computers
- AFTER SALES SERVICE – available on all computers out of guarantee
- VAT – all prices quoted above include VAT at 15%
- CREDIT FACILITIES – we offer credit over 12, 24 or 36 months, please ask for details

SILICA SHOP LIMITED
Dept PS1 1-4 The Mews, Hatherley Road, Sidcup,
Kent DA14 4DX Telephone 01-301 1111 or 01-309 1111